REVISTA COLOMBIANA DE COMPUTACIÓN
Volumen 11, número 2
Págs. 24 - 63

Using a RESTful messaging and registry system to support a range a distributed applications

Mark Baker*, Hugo Mills, Garry Smith, Matthew Grove, Rahim Lakhoo, Carl Albing

Fecha de Recibido: 05/07/2010 Fecha de Aprobación: 20/09/2010

Abstract

Tycho was conceived in 2003 in response to a need by the GridRM [1], a resource-monitoring project for a "light-weight", scalable and easy to use wide-area distributed registry and messaging system. Since Tycho's first release in 2006 a number of modifications have been made to the system to make it easier to use and more flexible. Since its inception, Tycho has been utilised across a number of application domains including wide-area resource monitoring, distributed queries across archival databases, providing services for the nodes of a Cray supercomputer, and as a system for transferring multi-terabyte scientific datasets across the Internet. This paper provides an overview of the initial Tycho system describes a number of applications that utilise Tycho, discusses a number of new utilities, and how the Tycho infrastructure has evolved in response to experience of building applications with it.

Keywords: *Restful, HTTP, PUT/GET, Transactions, Web 2.0, Peer-to Peer, Virtual Registry.*

* School of Systems Engineering, University of Reading, Reading, UK, E-mail: mark.baker@computer.org

1. Introduction

The Tycho project [2] focuses on the design and development of a system for binding together distributed applications with the aid of a combined messaging and registry system. Tycho provides application developers with a means of securely integrating distributed systems using a single software package. One aim is to simplify the process of assembling distributed applications by reducing the number of libraries or tools required by the application developer. A second aim is to produce a system that is more scalable and has higher performance than the combinations of registry and messaging software previously available.

Tycho has a Service-Oriented Architecture (SOA), where there are consumers, producers and mediators. The services provided by these components are:

- Mediators, which allow producers and consumers to discover each other and establish remote communications.
- Consumers typically subscribe to receive information or events from producers.
- Producers gather and publish information for consumers.

The design philosophy for Tycho was to keep its core relatively small, simple and efficient, so that it has a minimal memory foot-print, is easy to install, and is capable of providing robust and reliable services. More sophisticated services can then be built on this core and are provided via libraries and tools to applications. This enables Tycho to be flexible and extensible so that it will be possible to incorporate additional features and functionality.

In Tycho, producers and/or consumers publish their existence in a directory service known as the Virtual Registry (VR), which is a distributed peer-to-peer service provided by the network of mediators. A client uses the VR to locate other clients, which act as a source or sink for the data they are interested in. Normally, clients communicate directly, however, for clients that do not have direct access to the Internet, the mediator provides wide-area connectivity by acting as a gateway or proxy into a localised Tycho installation. The Tycho VR is made up of a collection of services that provides the management of client information and facilitates locating and querying remote Tycho installations.

A consumer (the client) registers with a local mediator, as part of the VR when it starts-up. The VR provides a locally unique name for each client

(e.g. a URI) and periodically checks registered entities to ensure their liveliness, the system removes stale entries if necessary. Tycho's architecture is designed to support both encryption and access control to provide a secure environment. Encryption is provided at the transport handler level using SSL to encrypt messages sent via the HTTP and Socket handlers.

1.1 RESTful Systems

The **Representational State Transfer (REST)** is a style of software architecture for distributed hypermedia systems such as the World Wide Web. The term Representational State Transfer was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation [3]. Fielding is one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification versions 1.0/1.1. REST defines a set of architectural principles that focuses on system resources, including how the resource state is addressed and transferred over HTTP by a wide range of clients, which can be written in different languages.

The REST-style architecture consists of clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of "representations" of "resources". A resource can be essentially any meaningful resource that may be addressed. A representation of a resource is typically a document that captures the current or intended state of a resource. At any particular time, a client can either be transitioning between application states. A client in a rest state is able to interact with its user, but creates no load and consumes no per-client storage on the set of servers or on the network. The client begins by sending requests, when it is ready to transition to a new state. While one or more requests are outstanding, the client is considered to be in a transitioning state.

REST did not attract much attention when Roy Fielding first introduced it in 2000. The analyses of this software architecture that uses the Web as a platform for distributed computing is now used a lot. Years after its introduction, a major frameworks for REST has appeared and is used because it provides a useful communication protocol, which is used via various systems, including Java JSR-311 [4].

The REST architectural style describes the following six constraints applied to the architecture, while leaving the implementation of the individual components:

- **Client–server:** The clients are separated from servers by a uniform interface. This separation of concerns means that clients are not

concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface is not altered.

- **Stateless:** The client–server communication is constrained by client context, which is stored on the server between requests. Each request from any client contains all of the information necessary to service the request, and any state is held in the client. This not only makes servers more visible for monitoring, but also makes it more reliable in the face of partial or network failures, as well as further enhancing system scalability.
- **Cacheable:** On the Web, clients are able to cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or not to prevent clients reusing stale or inappropriate data in response to further requests. Well-managed caching eliminates some client–server interactions, further improving scalability and performance.
- **Layered system:** A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load balancing and by providing shared caches. They may also enforce security policies.
- **Code on demand (optional):** Servers are able to temporarily extend or customise the functionality of a client by transferring logic to it that it can execute. Examples of this may include compiled components such as Java applets, AJAX, and client-side scripts, such as JavaScript.
- **Uniform interface:** The uniform interface between clients and servers simplifies and decouples the architecture, which enables each part to evolve independently.

Resource
Collection URI, such as http://example.com/resources/
Element URI, such as http://example.com/resources/MARK
GET
List the members of the collection, complete with their member URIs for further navigation. For example, list all the cars for sale.
Retrieve a representation of the addressed member of the collection expressed in an appropriate MIME type
PUT
Meaning defined as "replace the entire collection with another collection".
Update the addressed member of the collection or **create** it with the specified ID.

POST
Create a new entry in the collection where the ID is assigned automatically by the collection. The ID created is usually included as part of the data returned by this operation.
Treats the addressed member as a collection in its own right and creates a new subordinate of it.
DELETE
Meaning defined as "delete the entire collection".
Delete the addressed member of the collection.

Fig. 1: RESTful Service HTTP methods [5]

Figure 1 shows the characteristics of resources, GET, PUT, POST and DELETE. One of the key characteristics of a RESTful system is the explicit use of HTTP methods in a way that follows the protocol as defined by RFC 2616 [6]. HTTP GET, for instance, is defined as a data-producing method that intends to be used by a client application to retrieve a resource, to fetch data from a server, or to execute a query with the expectation that the server will look for and respond with a set of matching resources. REST uses HTTP methods explicitly and in a way it is consistent with the protocol definition. This basic REST design principle establishes a one-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods. According to this mapping:

- To create a resource on the server, use POST.
- To retrieve a resource, use GET.
- To change the state of a resource or to update it, use PUT.
- To remove or delete a resource, use DELETE.

The request URI in an HTTP GET request, for example, usually identifies one specific resource. Or the query string in a request a URI includes a set of parameters that defines the search criteria used by the server to find a set of matching resources. If the Web API uses GET to invoke remote procedures, it looks like this in Table 1.

```
GET /adduser?name=Mark HTTP/1.1
```

Table 1: The use of GET

If successfully processed, the result of the request is to add a new user, in this example, Mark - to the underlying data store. The problem here is mainly semantic. Web servers are designed to respond to HTTP GET requests by retrieving resources that match the path (or the query criteria) in the request URI. They return these or a representation in a response, which may not add a record to a database. From the standpoint of the intended use of the protocol method, and from the

standpoint of HTTP/1.1 - servers use GET in a inconsistent way. Beyond the semantics, the other problem with GET is to trigger the deletion, modification, or addition of a record in a database, or to change server-side state in some way. This is where it invites Web caching tools and search engines to make server-side changes simple by crawling a link. A simple way to overcome this common problem is to move the parameter names and values on the request URI into XML tags. The resulting tags, an XML representation of the entity to create, may be sent in the body of an HTTP POST whose request URI is the intended parent of the entity (see Table 1 and Table 2).

```
GET /adduser?name=Mark HTTP/1.1
```

Table 2: Listing 1 – Before

```
POST /users HTTP/1.1
Host: myserver
Content-Type:
application/xml
<?xml version="1.0"?>
<user>
<name>Mark</name>
</user>
```

Table 3: Listing - After

The methods shown in Table 2 and Table 3 describe a RESTful request. The use of HTTP POST shows the payload in the body of the request. On the receiving end, the request may be processed by adding a resource contained in the body as a subordinate of the resource identified in the request URI; in this case the new resource should be added as a child of /users. This relationship between the new entity and its parent, as specified in the POST request, is analogous to the way a file is subordinate to its parent directory. The client sets up the relationship between the entity and its parent; it defines the new entity's URI in the POST request. A client application may then get a representation of the resource using the new URI, noting that at least logically the resource is located under /users, as shown in Table 4.

```
GET /users/Robert HTTP/1.1 Host: myserver Accept: application/xml
```

Table 4: HTTP GEt quest

Using GET in this way is explicit because GET is for data retrieval only. GET is an operation that should be free of side effects, a property also known as idempotence. A similar refactoring of a Web method also

needs to be applied in cases where an update operation is supported over HTTP GET, as shown in Table 5.

```
GET /updateuser?name=Mark&newname=Bob HTTP/1.1
```

Table 5: Update over HTTP GET

This changes of the name attribute (or property) of the resource. While the query string can be used for such an operation, and Table 5 is a simple one, this query-string-as-method-signature pattern tends to break down when used for more sophisticated operations. Because the goal is to make explicit use of HTTP methods, a more RESTful approach is to send an HTTP PUT request to update the resource, instead of HTTP GET, for the same reasons stated above (see Table 6).

```
PUT /users/Mark HTTP/1.1
Host: myserver
Content-Type: application/xml
<?xml version="1.0"?>
<user> <name>Bob</name> </user>
```

Table 6: HTTP PUT Request

Using PUT to replace the original resource provides a much cleaner interface that is consistent with REST principles and with the definition of HTTP methods. The PUT request in Table 6 is explicit, in the sense that it points at the resource to be updated. It identifies it in the request URI and in the sense that it transfers a new representation of the resource from client to server in the body of a PUT request instead of transferring the resource attributes as a loose set of parameter names and values on the request URI. Table 6 also has the effect of renaming the resource from Mark to Bob, and in doing so changes its URI to /users/Bob. In a REST service, subsequent requests for the resource using the old URI would generate a standard 404 Not Found error. As a general design principle, it helps to follow REST guidelines for using HTTP methods explicitly by using nouns in URIs instead of verbs. In a RESTful service, the verbs - POST, GET, PUT, and DELETE - are already defined by the protocol. Ideally, to keep the interface generalised and to allow clients to be explicit about the operations they invoke, the service that should not define more verbs or remote procedures, such as /adduser or /updateuser. This general design principle also applies to the body of an HTTP request, which is used to transfer the resource state, but not to carry the name of a remote method or remote procedure to be invoked.

RESTful services need to scale to meet increasingly high performance demands. Clusters of servers with load-balancing and failover capabilities, proxies, and gateways are typically arranged in a way that forms a service

topology, which allows requests to be forwarded from one server to another, as needed to decrease the overall response time of a service call. Using intermediary servers to improve scale; which requires a RESTful service client to send complete, independent requests. That is, to send requests that include all data needed to be fulfilled so that the components in the intermediary servers may forward, route, and load-balance without any state being held locally in between requests. A complete, independent request does not require the server, while processing the request, to retrieve any kind of application context or state. A RESTful service application (or client) includes within the HTTP headers and body of a request all of the parameters, context, and data needed by the server-side component to generate a response. Statelessness in this sense improves service performance and simplifies the design and implementation of server-side components because the absence of state on the server, which removes the need to synchronise session data with an external application.

Figure 2 illustrates a stateful service, which an application may request the next page in a multi-page result set, assuming that the service keeps track of where the application leaves off while navigating the set. In this stateful design, the service increments and stores a previousPage variable somewhere to be able to respond to requests for next.

Fig. 2: Stateful Design

The stateless server-side components are less complicated to design, write, and distribute across load-balanced servers. A stateless service not only performs better, it shifts most of the responsibility of maintaining state to the client application. In a RESTful service, the server is responsible for generating responses and for providing an interface that enables the client to maintain application state on its own. For example, in the request for a multi-page result set, the client should include the actual page number to retrieve instead of simply asking for next (see Figure 3).

Fig. 3: Stateless Design

A stateless service generates a response that links to the next page number in the set, and lets the client do what it needs to do, in order to keep this value around. This aspect of RESTful service design can be broken down into two sets of responsibilities, as a high-level separation that clarifies just how a stateless service can be maintained.

The server generates responses that include links to other resources, which allow applications to navigate between related resources. This type of response embeds links. Similarly, if the request is for a parent or container resource, then a typical RESTful response might also include links to the parent's children or subordinate resources, so that these remain connected. It also generates responses that indicate whether they are cacheable or not to improve performance by reducing the number of requests for duplicate resources and by eliminating some requests entirely. The server does this by including a Cache-Control and Last-Modified (a date value) via an HTTP response header.

The client application uses the cache-control response header to determine whether to cache the resource (make a local copy of it) or not. The client also reads the Last-Modified response header and sends back the date value in an If-Modified-Since header to ask the server if the resource has changed. This is called Conditional GET, and the two headers go hand-in-hand so that the server's response is a standard 304 code (Not Modified) and omits the actual resource requested if it has not changed since that time. A 304 HTTP response code means the client can safely use a cached, local copy of the resource representation as the most up-to-date, in effect bypassing subsequent GET requests until the resource changes. It sends complete requests that can be serviced independently of other requests. This requires the client to make full use of HTTP headers as specified by the service interface, and to send complete representations of resources in the request body. The client sends requests that make very few assumptions about prior requests, the existence of a session on the server. The ability of the server to add a context to a request, or regarding the application state, is kept in between requests. This collaboration between client application and service is essential to being stateless in a RESTful service. It improves performance by saving bandwidth and minimising server-side application state.

1.2 Exposure of directory structure-like URIs

From the standpoint of client application addressing resources, the URIs determines how intuitive the REST service is going to be, and whether the service is going to be used in ways that the designers can anticipate. A third RESTful service characteristic is all about the URIs. The RESTful service URIs should be intuitive to the point

where they are easy to understand. One thinks of a URI as a kind of self-documenting interface that requires little, if any, explanation or reference for a developer to understand what it points to and to derive related resources. To this end, the structure of a URI should be straightforward, predictable, and easily understood. One way to achieve this level of usability is to define directory structure-like URIs. This type of URI is hierarchical, rooted at a single path, and branching from it are sub-paths that expose a service's main areas. According to this definition, a URI is not merely a slash-delimited string, but rather a tree with subordinate and super-ordinate branches connected at nodes. For example, you might define a structured set of URIs like this:

http://www.myservice.org/discussion/topics/{topic}

The root, /discussion, has a /topics node beneath it. Underneath that there are a series of topic names, such as gossip, technologies, and so on. In some cases, the path to a resource lends itself especially well to a directory-like structure. The resources organised by date, for instance, which are a match for using a hierarchical syntax. This example is intuitive because it is based on rules:

http://www.myservice.org/discussion/2008/12/10/{topic}

The first path fragment is a four-digit year, the second path fragment is a two-digit day, and the third fragment is a two-digit month. It may seem a little complicated to explain, but this is the level of simplicity we are after. Humans and machines can easily generate structured URIs like this because they are based on rules. Filling in the path parts in the slots of a syntax makes them good because there is a definite pattern from which to compose them:

http://www.myservice.org/discussion/{year}/{day}/{month}/{topic}

Some additional guidelines about a URI structure for a RESTful service are:

- Hide the server-side scripting technology file extensions if any, so you can port to something else without changing the URIs.
- Keep everything lowercase.
- Substitute spaces with hyphens or underscores (one or the other).
- Avoid query strings as much as you can.
- Instead of using the 404 Not Found code, if the request URI is for a partial path, always provide a default page or resource as a response.

URIs should also be static, so that when the resource changes or the implementation of the service changes, the link stays the same. This allows bookmarking. It is important that the relationship between resources, are encoded in the URI, which remains independent of the way the relationships are represented where they are stored.

As examined in the so-called principles of RESTful interface design, XML over HTTP is a interface that allows internal applications, such as Asynchronous JavaScript + XML (AJAX)-based custom user interfaces, to easily connect, address, and consume resources. In fact, the fit between AJAX and REST has increased the amount of attention about REST these days. Exposing a system's resources through a RESTful API is a flexible way to provide different kinds of applications with data formatted in a standard way. It helps to meet integration requirements that are critical to building systems, where data can be easily combined (mashups) and to extend or build a set of base, where RESTful services are much bigger.

1.3 State of the Art.

Peer-to-peer architectures have been employed for a variety of different application categories, which include the following:

- **Communication and Collaboration:** This category includes systems that provide the infrastructure for facilitating direct, usually real-time, communication and collaboration between peer computers. Examples include chat and instant messaging applications, such as Chat/IRC Instant Messaging (Skype, AOL, ICQ, Yahoo, and MSN), and Jabber [7].
- **Distributed Computation:** This category includes systems whose aim is to take advantage of the available peer computer processing power. This is achieved by breaking down computer-intensive tasks into small work units and distributing them to different peer computers, which execute their corresponding work unit and return the results. Central coordination is invariably required, mainly for breaking up and distributing the tasks and collecting the results. Examples of such systems include projects such as Seti@home [11][13], genome@home [14][16], and others.
- **Internet Service Support:** A number of different applications based on peer-to-peer infrastructures have emerged for supporting a variety of Internet services. Examples of such applications include peer-to-peer multicast systems [10][11], Internet indirection infrastructures [12], and security applications providing protection against denial of service or virus attacks [17][18][19].

- **Database Systems:** Considerable work has been done on designing distributed database systems based on peer-to-peer infrastructures. Bernstein et al. [20] propose the Local Relational Model (LRM), in which the set of all data stored in a peer-to-peer network is assumed to be comprised of inconsistent local relational databases interconnected by sets of "acquaintances" that define translation rules and semantic dependencies between them. PIER [14] is a scalable distributed query engine built on top of a peer-to-peer overlay network topology that allows relational queries to run across thousands of computers. The Piazza system [22] provides an infrastructure for building Semantic Web [23] applications consisting of nodes that can supply source data (e.g. from a relational database), schemas (or ontologies) or both. Piazza nodes are transitively connected by chains of mappings between pairs of nodes, allowing queries to be distributed across the Piazza network. Finally Edutella [24] is an open-source project that builds on the W3C metadata standard Resource Description Framework (RDF), to provide a metadata infrastructure and querying capability for peer-to-peer applications.
- **Content Distribution:** Most of the current peer-to-peer systems fall within the category of content distribution, which includes systems and infrastructures designed for the sharing of digital media and other data between users. Peer-to-peer content distribution systems range from relatively simple direct file sharing applications, to more sophisticated systems that create a distributed storage medium for securely and efficiently publishing, organising, indexing, searching, updating, and retrieving data. There are numerous systems and infrastructures, some examples of which are: Napster [25], Publius [26], Gnutella [27], Kazaa [28], Freenet [29], MojoNation [30], Oceanstore [31], PAST [33], Chord [34], Scan [36], FreeHaven [37], Groove [38], and Mnemosyne [39].

A distributed hash table (DHT) framework, allows a peer-to-peer system to maintains a global key-to-document index. The approach is characterised by the following ideas. It can carefully select the indexing keys so that they consist of terms and sets that are discriminative with respect to a global document collection. They appear in a limited number of documents, thus limiting the size of the posting lists associated with the keys in the global inverted index. This directly addresses the main problem identified in for structured peer-to-peer Web search, namely the processing and transmission of extremely large posting lists. The achieved key-based indexing procedure results in the retrieval because the posting lists associated with the keys in the global index are short and correspond to pre-computed and therefore readily available results for potential multiple term queries. The advance of

Tycho, over a DHT, is that it has a virtual registry that can hold not only metadata, but also potentially Semantic Web information. So, this means that rather than having distributed hash table, it is much easier to search across Tycho to find a peer that has the correct data and information that may be needed.

1.4 Tycho Implementation

Tycho is a Peer-to-Peer (P2P) framework for developing distributed applications. It consists of a core communications component, the mediator, and a lightweight client API, which provides an asynchronous interface for sending messages to other clients via the mediator interface. The mediators contain a distributed database, based on Hypersonic, which can be queried as a single entity.

Tycho's implementation is based on REST [3] in order to provide standardised services without the need to follow long and complicated Web Services or Grid standards that were and are constantly in flux. Tycho was designed to be easy to deploy and as a result only requires an installed Java Virtual Machine (JVM) in order to execute the single Java JAR file that consists of the complete Tycho download (all Tycho dependencies such as the mediator's internal database and communications libraries for the virtual registry (VR) are included within this file).

Tycho was intentionally designed around a minimal core that only provides essential services. The idea is that the core should be fast and unencumbered by features that are not essential to Tycho's roles as a distributed registry and messaging system. This approach differs from the method used by other middleware, such as NaradaBrokering (NB) [42] and the Globus Monitoring and Discovery System [43], where new functionality is typically added to the core implementation. In Tycho, new functionality is packaged as optional utilities that sit above the core services.

1.5 The Structure of a Tycho network

The core of a Tycho communications network consists of one or more mediators (see the diamonds marked M in Figure 4). Each mediator knows the addresses of all of the other mediators, through the action of a boot-strapper mechanism. It is the job of the mediators to distribute queries, commands, replies and error reports to the appropriate destination(s) within the network. Communication between mediators is by means of HTTP or HTTPS traffic, running (by convention) on TCP using port 8080.

Each mediator may have any number of Tycho clients associated with it (see the circles marked C in Figure 4). A client communicates to the rest of the Tycho network by sending messages to its local mediator. The mediator can then decide where to send the message – to a specific mediator, or as a broadcast to all mediators. Replies to the message are routed back to the client. Typically, the client - mediator communications are across raw TCP sockets, or using SSL/TLS sockets.

Finally, each mediator also maintains a virtual registry component, which is a simple distributed database (see the squares marked in VR in Figure 4), containing basic information about every client registered with that mediator. One of the fields contained in the virtual registry is "client data" field that can be filled with any [text] data the client desires. This can be used to store additional data or metadata about the services that the client is offering to the network.

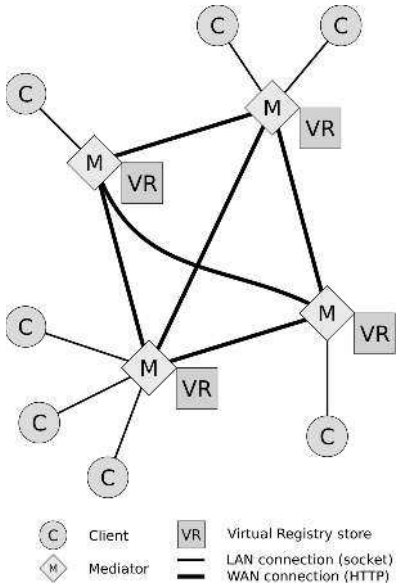

Fig. 4 : Top-Level View of Tycho

Since synchronous communications typically have a higher overhead than asynchronous (due to the need to track messages and the time spent blocking and waiting for responses) a decision was made early in Tycho's development that its core should only support asynchronous communication patterns. This means that applications must either be written following an asynchronous programming model, or utilise a higher-level blocking API potentially implemented as a Tycho utility.

1.6 Networking protocol

The Tycho network protocol consists of two logical layers:

- A transport layer, which governs the structure of the data packets exchanged between members of the Tycho network.
- The control layer specifies a set of message types and interactions that implement the functionality of a Tycho network.

The transport layer is very simple, consisting of single data packets passed between the client and mediator, or between two mediators in the core of the network. For the client, each mediator message is sent as "raw" data via a TCP socket interface. For mediator messages, the basic Tycho protocol message is wrapped up as data in an HTTP request, the HTTP response being simply an indicator that the message was successfully received and parsed.

Position (Octets)	Length (Octets)	Description
0	4	*m* = Complete message length (in octets)
4	4	Sequence number of this message
8	4	Command number
12	4	*f* = From length (in characters)
16	4	*t* = To length (in characters)
20	4	*p* = Payload length*
24	2*f*	The URL of the source of the message, in UTF16-BE form.
24+2*f*	2*t*	The URL of the destination of the message, in UTF16-BE form.
24+2*f*+2*t*	*p* or 2*p*†	The payload of the message.

Table 7: The Structure of the Tycho Packet Protocol

* The payload length may be either the count of string length (if the payload is internally a string), or the number of octets (if the payload is internally a byte array). This ambiguity is a primary reason for the development of the V2 transport protocol mentioned in section *n*, below.

† The actual length of the payload in octets, and its content type, can be determined from the overall message length, *m* (at offset 0 of the header), and the values of *f*, *t* and *p*.
If $m = 24 + 2f + 2t + 2p$, then the content type is String, and the length of the payload in octets is $2p$.
If $m = 24 + 2f + 2t + p$, then the content type is an array of octets, and the length of the payload in octets is simply p.
In any other case, the header is probably malformed, and should be ignored.

The structure of a Tycho protocol packet is shown in Table 7. All numbers are encoded in big-endian form ("network order"), and all strings are encoded as big-endian UTF-16. The use of UTF-16 means that where string lengths appear (e.g. the f and t values), the length of the data in octets is twice the length of the string, as each character is encoded as two octets.

Code	Name	Description
1	Ping	Ping request. Sent from mediator to client.
101	Pong	Ping reply. Sent from client to mediator.
11	Register	Client registration with mediator and VR.
12	Unregister	Client un-registration from mediator and VR.
15	Update VR	Client update of virtual registry with new details.
115	Update reply	Response from VR to indicate success.
13	VR Query	Request from client to mediator to query the VR.
33	VR Query	Mediator-to-mediator VR query made on behalf of a client.
133	VR response	Response to a message 33 query, which is returned to a mediator.
113	VR response	Response to a message 13 query, which is returned to the originating client.
-1	Error	An error occurred within the Tycho network. This message indicates the fault type.
21	Message	A message intended for receipt by another client in the network. Payload is application-specific.

Table 8: The list of Tycho commands

In general, the command number for a response to a message with command number n is n+100. A message forwarded from one mediator to another on behalf of a client has command number n+20.

1.7 Security

Security is an essential requirement for any distributed system. Tycho's architecture is designed to support both encryption and access control to provide a secure environment. Encryption is provided at the transport handler level using SSL to encrypt messages sent via the HTTP, Socket

handlers. Access control is provided using a layered approach. In keeping with the design philosophy of Tycho, we use existing infrastructure. Access control is can be via the use of a proxy server, or the security features of a firewall. For instance, when deploying Tycho on a cluster, a common configuration uses a proxy server on the head node to control access to mediators running on compute nodes. An alternative mechanism for access control is provided by a pluggable authentication library, which could interface with existing security protocols and solutions such as WS-Security or the Java Authentication and Authorization Service (JAAS).

1.8 Tycho Client Internals

The Tycho client interface is deliberately very simple. The API consists of a single Java interface, which must be implemented by any Tycho client to receive messages, plus a connector object used to construct and send messages. Helper classes for managing timeouts and embedding a mediator within the client are also supplied. Here we briefly describe the main methods of the Client API:

- **Class TychoConnector:** provides the core client functionality. It contains fourteen methods that provide a high-level API for the Tycho functionality. The API handles interaction with other clients and the Tycho mediator transparently. For example, to perform a distributed Tycho query across the entire VR a single method is used with one parameter.
- **Class EmbeddedMediator:** starts an instance of a Tycho mediator within the same JVM as the client. It provides an additional three methods to retrieve settings from the mediator (such as its URL) and allows the verbosity of the debug output to be set.
- **Class Message:** contains fourteen methods for manipulating Tycho messages. It provides a high-level interface for users to read and alter parts of a binary message without having to directly manipulate the binary data.
- **Finally a single Java interface is provided - Interface IeventInterface:** which defines five methods that are implemented by all Tycho clients. It allows asynchronous messages to be delivered to the client.

A Tycho client must create a TychoConnector object, which connects to a specific mediator (i.e. to a specific IP address and port number). The TychoConnector offers methods for constructing and sending messages to other parts of the Tycho network, plus methods for updating and querying the virtual registry.

Tycho is an asynchronous system: when a message is sent via the TychoConnector object, control is returned immediately to the sender, which must wait for the reply or replies to come back to it at some later time. When a message is sent, the sender is given a message ID, which is unique within that client instance, and that can be used to identify incoming replies marked with the same ID.

To receive messages, the client must implement the IEventInterface interface. This consists of five methods which are called when specific types of message or event is received: Initial registration, VR update of records held on behalf of this client, responses to a VR query, messages from other clients, and error messages. The IEventInterface is thus implemented as a construction parameter for the TychoConnector described above.

Behind the client API, there is a simple server, running in its own thread, which listens for incoming connections on a TCP socket, reads the message sent by each connection, and injects that message into a blocking message queue. The main thread of the connector then takes messages out of the queue and calls the appropriate method of the IEventInterface to pass the message to application control. The latest Tycho APIs can be found on this URL (http://acet.reading.ac.uk/bin/projects/tycho/software/1.2/doc/user/). The APIs have been changed and simplified as we have worked on various applications.

1.9 Tycho Mediator Internals

The Tycho mediator is very simple from an API point of view. It is used to start a Tycho service. However, internally a mediator is considerably more sophisticated than the client-side software. The general structure of a mediator is shown in Figure 5.

The primary function of a mediator is to route incoming messages to their intended destination. Incoming messages enter the mediator, either from other mediators (via HTTP or HTTPS), or from clients (via raw sockets, TLS, or a shared memory mechanism in the case of an embedded mediator). The incoming messages are placed in a queue, and processed in sequence by a CommsManager thread, which inspects the command number of each message to decide where that message should be routed to, and forwards it on. The possible internal destinations for messages are:

- An internal ping processor,
- The local part of the VR store,
- A dispatcher to forward queries on to other mediators
- Local clients,

- A handler for forwarding returned responses from other mediators or clients.

The internal ping processor (ClientPinger) simply returns a "pong" message immediately to the originating component, such as the client or mediator. This mechanism can be used to ensure that the local list of mediators and clients is kept up to date, by removing non-responsive peers.

The local part of the virtual registry (the LocalStore) is simply an SQL database to which virtual registry queries are first sent. If the query specified a limit to the number of records to be returned, and that limit is reached from the local store, the query stops immediately. If that limit is not reached, or none was given, the VR core will also pass on the query to all of the other mediators in the network to find additional records. Currently, three SQL database engines are supported: MySQL, Hypersonic SQL, and H2. The latter two are embedded within the mediator, and thus function as a single stand-alone process.

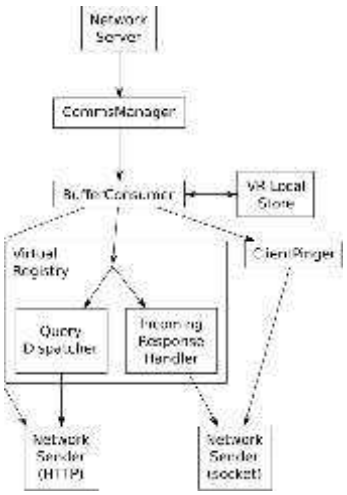

Fig. 5: Tycho Mediator Internals

Finally, messages may be received by a mediator for forwarding on to other components of the system. If a message is not a part of a VR transaction or a ping message, it is either sent directly to the client, in the case that the destination client is registered with this mediator, or sent on to the relevant mediator. Figure 5 shows the high-level architecture of Tycho, where there are consumers and producers

discovering each other and communicating via a LAN and across the Internet.

2 Tycho Plugins

Tycho provides a plug-in mechanism that allows different technologies to be used for the VR communications. The reference implementation provided HTTPS and Internet Relay Chat (IRC) [44] support. IRC networks provide a communications overlay that is configured to provide fault tolerant messaging. Tycho allows applications developers to make use of this existing Internet infrastructure without the need to deploy their own messaging services, daemons, or even change the configuration of Tycho from the default options.

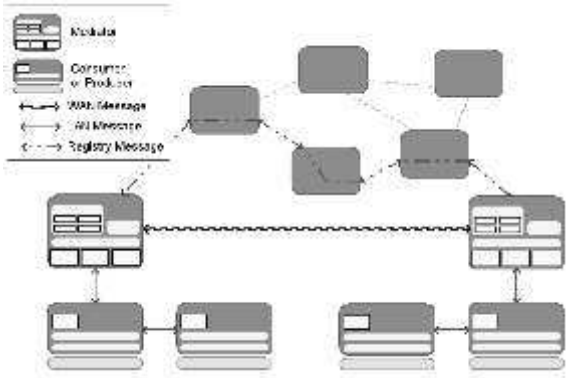

Fig. 6: The high-level architecture of Tycho, showing consumers and producers discovering each other and communicating via a LAN and across the Internet.

2.1 Tycho's Performance

When using a distributed registry to store data for an application, one task was to configure the registry to handle the necessary data. For example to configure Globus MDS [43] and the Relational Grid Monitoring Architecture (R-GMA) [45] to accept user-defined application specific data, all instances of R-GMA or MDS must be configured with the same data schema. In contrast, Tycho allows the clients to dynamically describe the data they publish into the VR by using the schema field, which means that VRs do not need to be reconfigured. Both the automatic configuration and the ability to dynamically publish ad-hoc information into the VR, reduces the administrative overheads of deploying and using Tycho, compared to these other systems.

Unlike related systems, such as NaradaBroker (NB), MDS, R-GMA, Jini [46] and Apache Axis [47], Tycho is distributed with security enabled by default and does not require the developer to install additional packages. The reference implementation of Tycho supports transport-level encryption using HTTPS and access control (using MD5 hashes), which prevents access to the Virtual Registry. Furthermore, these systems have multiple software dependencies that must be satisfied as part of the installation process; Tycho on the other hand only requires a JVM to operate.

Related systems require some manual configuration before they could be bootstrapped. They also need additional effort to arrange their components into a scalable hierarchy. The Tycho mediator automatically discovers and connects to other Tycho instances by using the bootstrapping functionality of the Virtual Registry (VR)-interconnects.
Various performance tests have been made to measured registry's performance and capability of Tycho, against MDS, and R-GMA. This revealed that Tycho had better performance than the other systems. During these tests the other systems failed due to memory management issues. As part of a peer-to-peer file-sharing test, Tycho successfully transferred files of up to 80 Gbytes to 60 peers.

2.2 Tycho Utilities and Applications

Tycho was first released in 2006, since then a number of updates to the software have been made to make it more functional, so that it can support a greater number of distributed applications. This section first briefly describes the range of applications that use Tycho, and then we discuss extensions to Tycho that have been made to better support a range of applications.

2.3 Cross-Database Search - XDB

The UK JISC-funded VERA project [48] investigated the development of a virtual environment for research in archaeology. The project was based around the archaeological excavation at Silchester, a Roman town that was abandoned in the fifth century. The excavation has been running for twelve years, and has accumulated a large database of information, stored in the IADB (Integrated Archaeological Database). One challenge within the project was to provide integrated search facilities across multiple archival databases.

The test system searches for records from the Silchester IADB, held in an ordinary relational database, and a classics-oriented collection of Roman-era inscriptions found in Vindolanda database [49], which is based on a RDF store. The cross-database search engine, XDB [50], see

Figure 7, was developed to investigate some of the issues arising from searching across multiple and highly disparate databases.

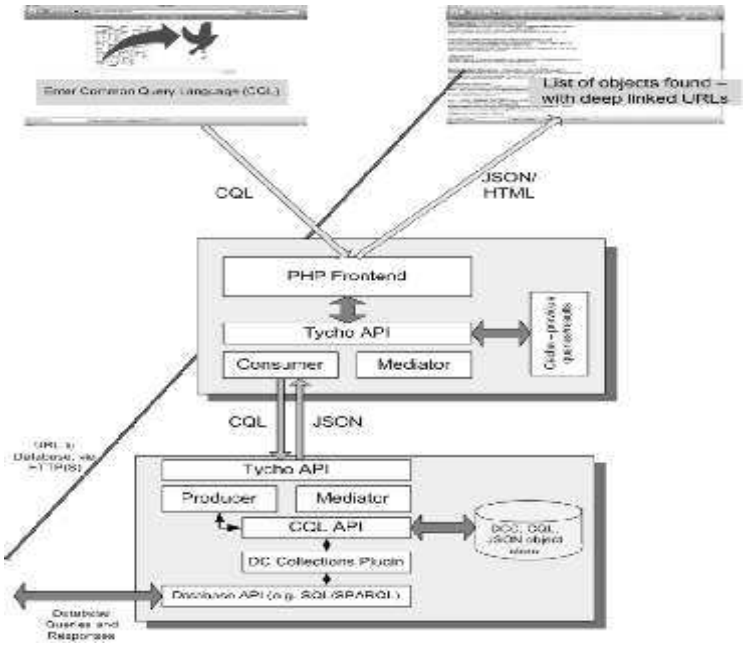

Fig. 7: The XBD Architecture

2.4 DNWay

The DNway project is attempting to create a generic framework that uses a master-worker paradigm to distribute work (idempotent tasks) across the computational resources of very large supercomputers and clusters. DNway is shown in Figure 8, which provides immediate, not queued, access to compute cycles and therefore must be:

- Adaptable - using processors and networks of varying speeds,
- Robust - adapt to changing response times, including the failure of remote workers,
- Accessible through firewalls,
- Able to cope with different network topographies.

In order to use DNway, the "work units" that are to be distributed must be defined, and the logic that will be used to process each unit of work must be written. Resource discovery, work distribution, result delivery, timeouts, and retry mechanisms are built on top of Tycho.

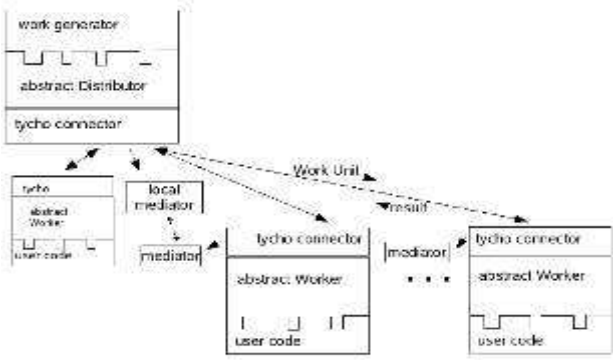

Fig. 8: The DNway Architecture

2.5 Necho

The Necho project [51] is creating a multi-tiered peer-to-peer system, which is akin to BitTorrent, for distributing multi-terabyte scientific datasets across the Internet. The concepts for this project first appeared when working with the Sloan Digital Sky Survey [52], where it was necessary to download the original dataset and use a modified version of WGET [53] to split it up, transfer the chunks across the Internet, then join the chunks together and update the database.

The Necho architecture is shown in Figure 9. It consists of a hierarchical Peer-to-Peer (P2P) system that is based around shared-portal services and unique peers donated by participating individuals and organisations. The goal of the project is to combine P2P, using volunteer computing and social networks to provide a way to distribute, contribute to, and manage very large datasets. Necho uses Tycho to distribute, index and retrieve chunks of data that from the original each overall dataset. We are currently testing Necho against other BitTorrent systems, such as Azureus [54], and our single tier version of Necho is proving to be much faster.

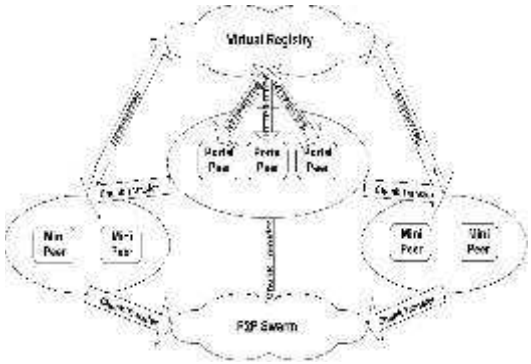

Fig. 9: A Schematic of the Necho Architecture

2.6 VOTechBroker

The VOTechBroker (VOTB) [55] is a system for submitting parameter sweeps to the Grid, and other distributed resources, in a transparent way. The VOTB aims to interoperate with a wide range of job submission systems using a plug-in component, and to protect the user from middleware details (see Figure 10).

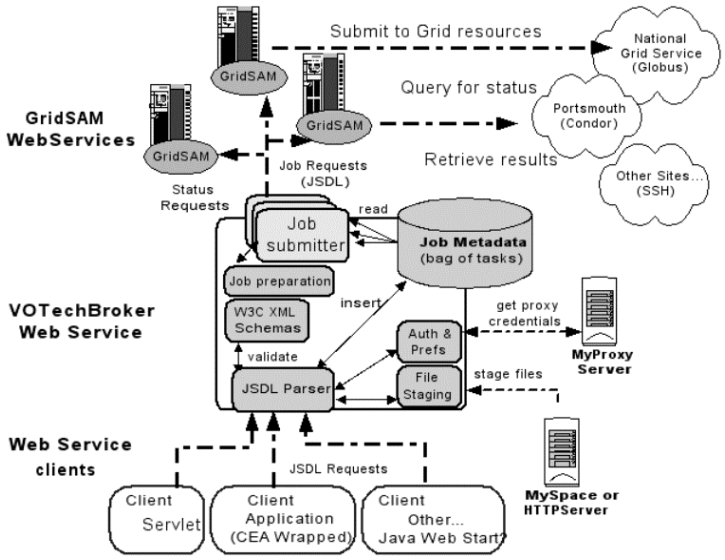

Fig. 10: The VOTechBroker Architecture

A key aim of the VOTB is to support an extensible range of (Grid) middleware, hide heterogeneity, and ease the complexity associated with job submission/execution. Tycho is used to dynamically locate available computational resources at remote sites that were appropriate to a particular user's requirements (e.g. with appropriate CPU architecture, libraries and services installed, account authorisation, availability, not over loaded, and sufficient free memory). This means that when a job is submitted to VOTB, the system can find the most appropriate system to schedule the task on.

2.7 GridRM

GridRM [20] is an extensible, wide-area, monitoring system that specialises in combining data from existing components and monitoring systems so that a consistent view of the underlying resources and services can be achieved, regardless of heterogeneity. Gateways (see Figure11) provide access to local resource information at each site. Clients connect to gateways to perform resource queries and to subscribe for events. GridRM uses Tycho in a number of ways to

bind together clients and Gateways for wide-area communications, and to provide the basis of an event mechanism (both wide-area events to clients, and events from local monitoring systems).

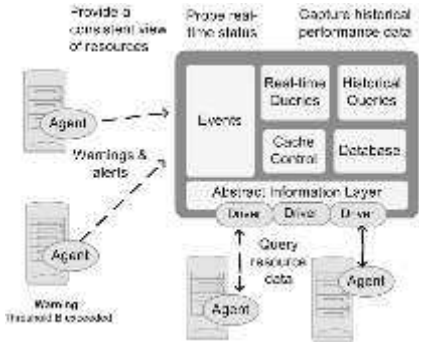

Fig. 11: The GridRM Architecture

2.8 The SORMA Project

The Self-organizing ICT Resource Management (SORMA) [57] is a European project developing methods and tools for efficient market-based allocation of resources. It uses a self-organising resource management system and market-driven models, which are supported by extensions to existing Grid computing infrastructure. Unlike many existing Grid environments, tasks submitted to SORMA are matched with available resources according to the economic preferences of both resource providers and consumers, and the current market conditions. This means that the classic job scheduler, which is based on performance rules, is replaced by a set of self-organising market-aware agents that negotiate service level agreements (SLAs), to determine resource allocation that best fulfils both performance and business goals. In SORMA, an economically enhanced resource manager (EERM) [21] exists at each resource provider's site and acts as a centralized resource allocator to support business goals and resource requirements.

The overall aim of the EERM is to isolate SORMA economic layers from the technical ones and orchestrate both economic and technical goals to achieve maximum economic profit and resource utilisation. The main goals of the EERM are:

- To combine technical and economic aspects of resource management,
- To perform resource price calculations, taking into account current market supply and demand,
- Create performance estimations, and business policies;
- To strengthen the economic feasibility of the Grid.

To provide a general solution that supports different scenarios and business policies, the EERM should provide flexibility in defining user (administrator) configurable rule-based policies, to support:

- Individual rationality: An important requirement for a system is that it is individually rational on both sides, i.e. both providers and clients have to have a benefit from using the system. This is a requirement for the whole system, including features such as client classification or dynamic pricing.
- Revenue maximisation: A key characteristic for SORMA providers is revenue (utility) maximization. The introduced mechanisms can indeed improve the utility of both the provider and the client.
- Incentive compatibility: Strategic behaviour of clients and providers can be prevented if a mechanism is incentive compatible. Incentive compatibility means that no other strategy results in a higher utility than reporting the true valuation.
- Efficiency: There are different types of efficiency. The first one considered here is Pareto efficiency: no participant can improve its utility without reducing the utility of another participant. Also the EERM must maximise the sum of individual utilities.

The EERM utilises GridRM to obtain resource information for system and per-process monitoring in order to determine if Service Level Agreements (SLAs) have been violated. The EERM is composed as a confederation of loosely bound components for scalability and availability reasons. Tycho provides messaging and event mechanisms within the SORMA system.

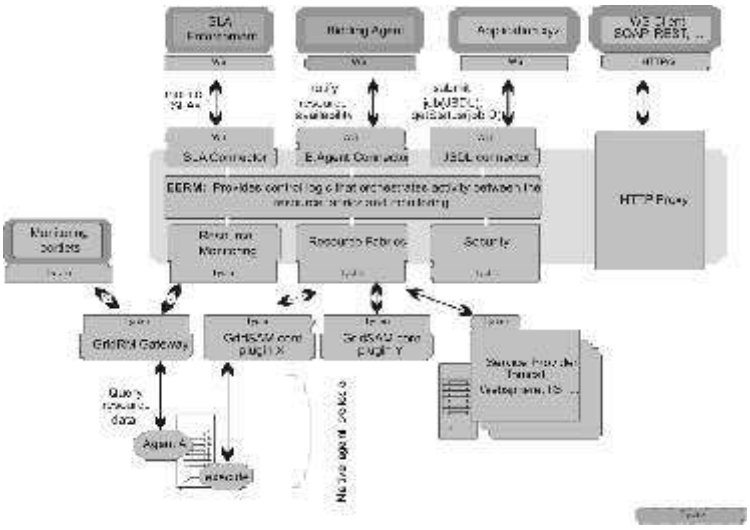

Fig. 12: The SORMA EERM Architecture

2.9 Slogger

Slogger [58] utilises various emerging Semantic Web technologies to gather data from heterogeneous log files generated by the various layers in a distributed system and unify them in common data store. The logs are ones generated by the operating system, middleware (e.g. Apache Tomcat or MPI) and applications themselves). Once unified, the log data can be queried and visualised in order to highlight potential problems or issues that may be occurring in the supporting software or the application itself. Slogger uses Tycho (see Figure 13) to first process, e.g. determine what data is need from the logs, and then gather data from the distributed resources and push this into a centralised RDF store. Once the data is in the store, SPARQL queries are issued in order analyse the RDF log data in order to identify problem and errors in the software executing over the distributed resources.

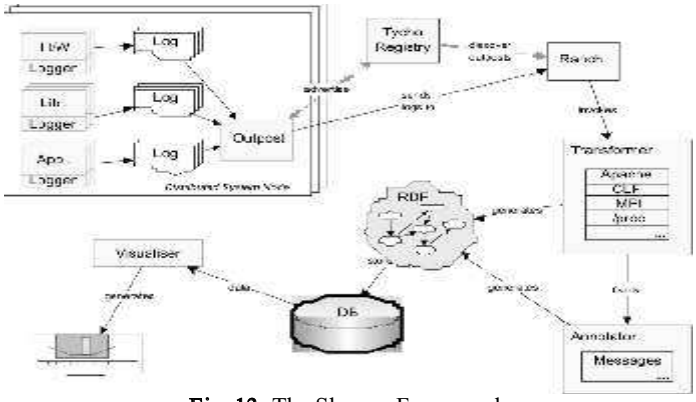

Fig. 13: The Slogger Framework

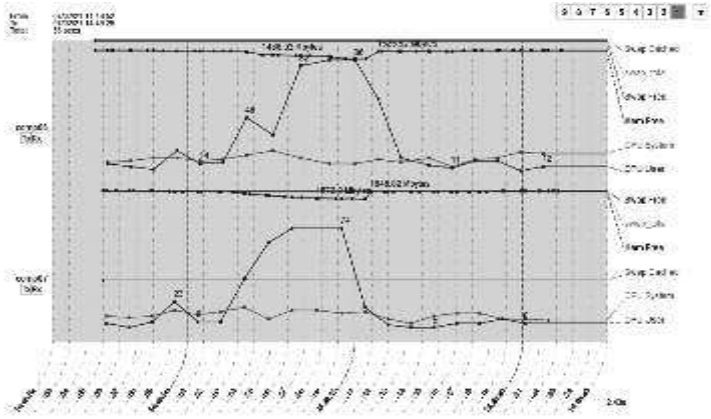

Fig. 14: A SVG Diagram showing CPU ad Memory Use

3 Map Service

Geographical maps are used to describe the Earth's surface and its contours. These maps are hosted on servers called "map servers" [59] around the globe and can be used by scientists in many ways. For example, to find the temperature of specific area, examine the wind pressure of particular place or study the state of oceans around the world. The environmental science community lacks a searchable registry of available Map Services. As a result, scientists cannot discover the data that may be most valuable for their work or they may spend a lot of time searching for the right data. Most of the time, the scientists manage to find the service, but this may be problematic and does not give access to data needed.

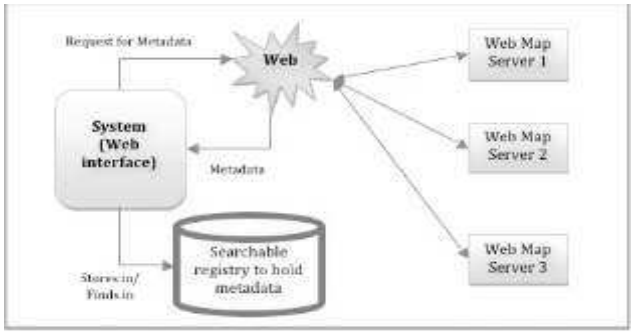

Fig. 15: The Map ServiceFramework

The Web contains a large number of valuable environmental datasets hosted on map servers. These datasets follow different specifications such as the Open Geospatial Consortium (OGC) Web Map Service. Users access these datasets to retrieve maps of desired functionality. The demand for maps has increased in recent years, especially with the environmental science community, who are facing difficulties in finding the right map service. As a result, scientists cannot discover the data that may be useful for their research work. Often, even if the scientist finds the correct map service, the service can be unreliable and fails to provide the required data. In this project, we have developed a framework that provides a searchable database for map services.

This project uses Tycho to search for map services, and find those that are most appropriate, based on the search data initiated by the user. A remote producer gathers keywords and metadata from each map server, and stores this into the VR; this data is collocated with the location of the server itself. When a user wants to find map data, it sends a query via

a consumer that is embedded into a Web browser; the query uses the OGC standards. The query is sent off, and first searches though the VR to find matching metadata or keywords, once this has been established, then the full OGC query is sent to the matching map server. The map server will then return the appropriate map to the client were it is displayed for the user.

3.1 Web 2.0 and Portlets

Tycho has been used with JSR-168 portlets in GridRM and SORMA. Furthermore, Web 2.0 interfaces that use Tycho to provide data to sliders have been implemented. JSR-168 portlets provide an opportunity to create user interfaces that are portable across different portal containers. We have created a number of JSR-168 portlets so that clients can remotely administer gateways and query resources from a Web browser. The portlets are currently hosted in a Gridsphere portal [31] and provide a modular approach for building a user interface; each portlet provides one type of functionality, and multiple portlets are combined (in the portal container) in order to provide the overall user interface. The portlets are categorised as those for performing gateway administration and those for querying resources and subscribing/receiving events. The portlets all utilise the client monitoring API (as a portlet service), which they use to communicate with gateways over the Global Layer. The front ends of the portlets are constructed using Java Server Pages (JSPs) that present XHTML controls and data to the client.

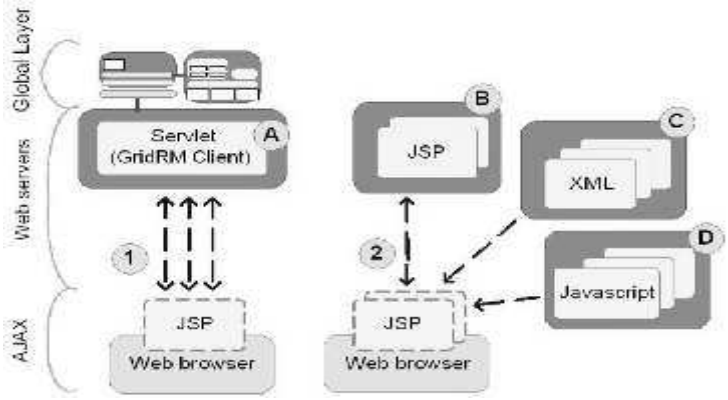

Fig. 16: The Structure of the Web 2.0 Interface

Although events are passed to the portlet code (via the monitoring API) in real-time, AJAX is required to refresh data in the portlet JSPs, so that events propagate to the user asynchronously. Alternatively the user is required to interact with the portlet user interface in order to be notified of new event data (e.g. by causing the portlet to enter its doView mode).

The structure of the Web 2.0 interface is shown on Figure 16.

Web 2.0 mashups provide an alternative way of displaying monitoring data. We have experimented with Web 2.0 to produce an interface that contains:

- Charts and gauges that display monitoring data dynamically in near real-time, for multiple resources registered with a gateway,
- A map that represents gateways according to their geographical location and displays their metadata, e.g. status, registered drivers, administrator information and network links between remote sites.
- A registry browser, that displays entries from Tycho's distributed registry in a graph and allows users to expand and collapse nodes as they browse registry meta data – this feature is useful to system developers and administrators.

It was interesting to compare and contrast the use of Flash-based Web 2.0 tools versus those based on simple technologies, which were less intrusive. We found that the Flash-based system contained a memory leak and would repeatedly crash the Web browser. Plus it was clear, even if the memory leak was solved, that if there were hundreds of resources being monitored, huge amount of CPU and memory would be used to monitor these resources.

As an alternative we looked at using simple slider icons, as shown in Figure 16, these were based on JavaScript and Cascading Style Sheets (CSS); they used small amount of memory and CPU, and could monitor hundreds of resource without being too intrusive.

4. Tycho Utilities

Utilities are software components that give the Tycho infrastructure greater functionality so that it can more robustly and reliably support distributed applications. In this section we describe some of the utilities that have been created.

4.1 Synchronous Monitoring API

By default, Tycho provides asynchronous messaging, however, many applications, such as the GridRM client requires a mix of blocking calls (e.g. for resource queries) and non-blocking calls (e.g. for event notifications). To achieve this we have created a Tycho utility that provides blocking communication operations. The Tycho method used by a client to transmit data is non-blocking and returns a unique message ID, which is used to key the semaphore into a hash table. When a message arrives at the client, the Tycho event handler passes the

message content to a method that matches the response to a request ID and performs a lookup in the hash table. If a match is found, a release is called on the semaphore and the blocked call continues. If a match is not found then the Tycho message is converted into an event of a given type and sent to the client's event listener. If the client has not subscribed to receive events of the given type, then the event is silently dropped.

A client API based on Tycho has been created that consists of seventeen calls for performing common operations such as:

- Registering and un-registering interest in receiving events from particular gateways and resources,
- Registering, un-registering and listing registered resources within a particular gateway,
- Querying "core" attribute values (e.g. memory utilisation, and system load),
- Controlling "resource watches", whereby a gateway is instructed to periodically capture resource data, which it saves to its internal database for later inspection,
- Controlling "job watches", whereby a gateway instructs a capable agent to monitor process resource utilisation at defined intervals. The gateway retrieves data from the agent and stores it in an internal database for later inspection.

4.2 HTTP pipelining

Currently every message sent by Tycho is via a separate HTTP message. In order to optimise the communication performance of Tycho, it requires us to open multiple parallel HTTP pipes, with buffering, when sending data to a remote destination, instead of reopening the single HTTP pipe every time. This project is still underway, but effectively provides the same functionality as GlobusFTP [41]. The system being developed has one HTTP control channel, and "N" parallel HTTP channels that are used to send data between Tycho components. This pipelining helps maximise communication performance between various end-points.

4.3 Lightweight Transactions

We needed to create lightweight transactions via the Tycho system, so that events sent around the system will be reliable and recoverable in case a failure occurs. Although transactions are valuable and provide atomicity, persistency, and recoverability, they are not widely used in programming environments today, due to their high overheads that have been driven by the latency of saving data to disks. A major challenge in transaction-based systems is to remove disk usage from the critical path of transactions. In this project, the so-called "lightweight transactions" will be created. Here there will be a transaction manager,

which uses main memory that decouples the performance of transactions from the disk.

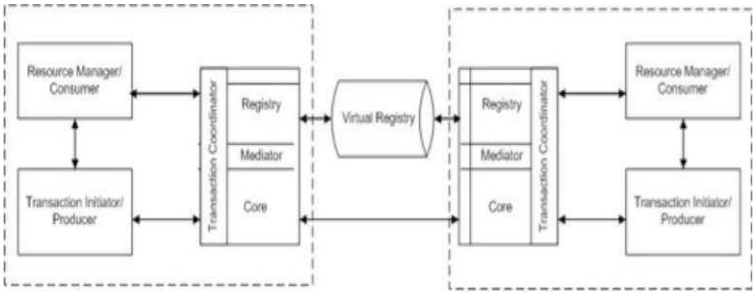

Fig. 17: The RESTful Transaction System

Figure 17 shows the RESTful transaction framework embedded in Tycho, where the Transaction Coordinator is embedded inside the Tycho Mediator. In this case Resource Managers and a Transaction Initiator could be with either the Producer or Consumer. They are connected to a Tycho Mediator and thus automatically connected to the Transaction Coordinator. The Mediator manages transactions along with the core functionality of the Tycho system and returns the result to the Transaction Initiator. Implementing the Transaction Coordinator inside the Tycho Mediator creates additional functionality to the core of the system, but results in slower performance. However, it allows logging in the internal Mediator database and as a result having a Log service and Transaction Coordinator in one node in the system, creates better persistence for all messages.

Once a distributed system maintains lightweight transactions, it can be used for various applications, such as computational steering or supporting events, when it is imperative to guarantee that a remote component receives the event. Principally, whenever an application is using multiple transactional and persistent resources, it may need distributed transactions.

4.4 Added VR Functionality

Our experience using Tycho's VR with various applications has shown us that it necessary to add greater functionality to the system. The original VR was designed to store the URLs of end points (producers or consumers) and also hold various XML documents that contained useful metadata related to the end-points. One extension for Tycho that came out of the XDB work is a results store, which manages the handling of query results from this interaction method in a thread-safe way. It does so by keeping track independently of results, which were "expected", and results, which were not. When the results store is told of

a message ID that has been sent, it checks the set of "unexpected" results, and matches up the ID with any result that may have come in.

The other obvious drawback of using Tycho for the XDB system is the limited storage space and search capability within the Tycho registry. Extending the registry to support arbitrary data tables (e.g. as in Necho) would make the implementation of the XDB's "master index" functionality much easier. Going even further than that, implementing storage of RDF [29] metadata within the registry, and searching through that distributed metadata using SPARQL [30] would be highly useful too. It is not immediately obvious to us how a full SPARQL implementation would work to find RDF fragments split across several mediators' registry stores, but even being able to search within each mediator's store for matching fragments could offer significant benefits to the XDB.

4.5 Additional Caching

One way to further improve performance is altering caching in the mediator to include local data-store queries in addition to remote responses. Adding indexing to the simple store would improve its performance when searching for records. In addition, the message-passing performance could be improved by changing the socket transport handler to use thread pooling to further reduce the cost of sending messages.

5. Summary and Conclusions

Tycho is a RESTful asynchronous messaging system with an integrated peer-to-peer virtual registry. Since, Tycho was first released at the end of 2006, we have increasingly used the system to support a range of distributed applications. We have used Tycho, rather than other systems, such as the Grid, because the RESTful services are easy to install and use. In addition, since Tycho was first designed, the overall standards used have not changed. Tycho uses HTTP (HTTPS) and Sockets (SSL) for communications. Internally, it uses SQL as the query language and uses LDAP LDIF to mark up responses from the VR. None of these standards have changed, and it ensures that applications, based on Tycho, will continue to work for the foreseeable future. Tycho's core is stable, but as we have pointed out in this paper, there are a number of features that need implementing to better support a wider range of applications. Some of the additional functionality needed by Tycho can be implemented by creating utilities and services on top of its generic API. An example of this is the Synchronous Monitoring API described earlier.

5.1 Future Work

A project is underway to develop the appropriate hardware and software to support remote monitoring of the environment via wireless sensor networks [27]. The project is using Sun SPOTs [28], which are small hardware platforms, battery operated, with a wireless device running the Squawk Java Virtual Machine (VM) without an underlying OS. This VM acts as both operating system and software application platform. We have been investigating the current software used, and we feel that using the Tycho producer/mediator on SPOT, and using HTPP communication will make the overall network more reliable and easier to program. For example, when the network starts up, each SPOT will gather data about all the SPOTs in the network. This information will be shown in the mediator and be used to calculate the optimal route to send result data back to the base station, and also, if there is a mote failure, it will be possible to calculate alternative routes back the base station.

Another project that we are considering is creating a system that provides "service mashups". Here Tycho components will be created, based on producers and consumers, which are registered in the VR. We will then build a graphical interface that can be used to discover and orchestrate the components together. An example of a useful service mashup could be some type of workflow system, where the various Tycho components would include parts of a workflow and they can be put together in a pipeline.

Bonjour, also known as zero-configuration networking, enables automatic discovery of computers, devices, and services on IP networks. Bonjour uses industry standard IP protocols to allow devices to automatically discover each other without the need to enter IP addresses or configure DNS servers. At present, a client must be given an explicit name or IP address and port number for talking to a local mediator. We plan on extending the client library to support Zeroconf networking protocols (mDNS and DNS-SD) to discover suitable mediators locally.

The current boot-strapper assumes a simple HTTP-based protocol for mediators to discover the other peers within a Tycho network. This introduces a single point of failure into the system, as the boot-strapper is required in order for the mediator to find its peers. Alternative mechanisms should be configurable at runtime. Possibilities include the use of global-area DNS-SD; IRC bots; PEX (peer exchange protocol, as used by BitTorrent); fixed sets of endpoints distributed through a file or URL; and self-bootstrapping (i.e. tell the mediator of the location of one other mediator, and it can obtain the full list of mediators in the network from there).

The VR implementation only maintains a single relational table within each mediator's local store, containing basic information about the clients connected to that mediator. It is possible for clients to store a limited amount of arbitrary textual data in a single field within that table, but more complex data relations are not feasible within the current framework. Allowing the creation of additional tables within the local store would offer much greater flexibility.

The VR is built around the use of SQL for its main functions, and thus assumes a traditional relational data model. However, there are many applications for which other data models are potentially more useful. For example, having the ability to store client metadata in RDF in the VR would assist the development of many applications. For example, in the Linksphere cross-database search facility, where multiple databases with incompatible schemas and metadata vocabularies are to be searched, the publication of RDF can be used to deal with the diverse sets and structures of the data.

Some of the bootstrap mechanisms use their own list of the mediators forming a given Tycho network. However, this can lead to very long start-up times for a mediator, if a high proportion of the entries in the list are stale, as the mediator attempts to contact each one in turn. Adding a timed leasing mechanism to the protocol would ensure that the set of stale entries is kept small.

5.2 Network performance

There are still a number of bottlenecks within the networking code of Tycho. Ameliorating these will require, amongst other things, a new version of the network protocol, laying out the packet headers differently, and a refactoring of the socket handling code to reduce the number of internal copies of message data.

References

[1] GridRM, http://gridrm.org

[2] Tycho, http://acet.rdg.ac.uk/projects/tycho/

[3] Roy Thomas Fielding. Architectural Styles and the Design of Network-based Software Architectures, Ph.D. Thesis. University of California, Irvine, Irvine, California, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/

[4] JSR 311 - https://jsr311.dev.java.net/

[5] RESTful Wiki - http://en.wikipedia.org/wiki/Representational_State_Transfer

[6] RFC 2616 - http://www.ietf.org/rfc/rfc2616.txt

[7] Jabber web site - http://www.jabber.org/

[8] W.T. Sullivan III, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A new major seti project based on project serendip data and 100,000 personal computers. In Proceedings of the 5th International Conference on Bioastronomy, 1997.

[9] K. Lai, M. Feldman, I. Stoica, and J Chuang, Incentives for cooperation in peer-to-peer networks. In Proceedings of the Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, June 2003.

[10] R. VanRenesse, K.P. Birman, A. Bozdog, D. Dimitriu, M. Singh, and W. Vogels, Heterogeneity-aware peer-to-peer multicast. In Proceedings of the 17th International Symposium on Distributed Computing (DISC 2003), Sorrento, Italy, October 2003.

[11] M. Castro, P. Druschel, A.M. Kermarree, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. IEEE Journal on Selected Areas in Communications, 20(8), October 2002.

[12] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In Proceedings of the ACM SIGCOMM'02 Conference, Pittsburgh, PA, August 2002.

[13] seti@home project web site. http://setiathome.ssl.berkeley.edu

[14] R. Huebsch, J.M. Hellerstein, N. Lanham, and B. Thau Loo. Querying the Internet with pier. In Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003.

[15] S.M. Larson, C. Snow, and V.S. Pande. Modern Methods in Computational Biology,, chapter Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. Horizon Press, 2003.

[16] genome@home project web site. http://genomeathome.stanford.edu/

[17] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. In Proceedings of the ACM SIGCOMM'02 Conference, Pittsburgh, PA, August 2002.

[18] R. Janakiraman, M. Waldvogel, and Q. Zhang. Indra: A peer-to-peer approach to network intrusion detection and prevention. In Proceedgings of 2003 IEEE WET ICE Workshop on Enterprize Security, Linz, Austria, June 2003.

[19] V. Vlachos, S. Androutsellis-Theotokis, and D. Spinellis. Security applications of peer-to-peer networks. Computer Networks Journal, 45(2):195-205, 2004.

[20] P. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu. Data management for peer-to-peer computing: A vision. In Proceedings of the Workshop on the Web and Databases, WebDB 2002.

[21] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks. In Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), MIT Faculty Club, Cambridge, MA, USA, March 2002.

[22] A.Y. Halevy, Z.G. Ives, P. Mork, and I. Tatarinov. Piazza: Data management infrastructure for semantic web applications. In Proceedings of the twelfth international conference on World Wide Web, pages 556-567, Budapest, Hungary, 2003.

[23] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. Scientific American, May 2001.

[24] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmer, and T. Risch. Edutella: A P2P networking infrastructure based on RDF. In Proceedings of the twelfth international conference on World Wide Web, Budapest, Hungary, 2003.

[25] Napster Web Site - http://www.napster.co.uk/

[26] M. Waldman, A.D. Rubin and L.F. Cranor. Publius: A robust, tamper-evident, censorship-resistant web publishing system. In Proceedings of the 9th USENIX Security Symposium, August 2000.

[27] Gnutella web site - http://gnutella.wego.com

[28] Kazaa web site - http://www.kazaa.com

[29] I. Clarke, O. Sandberg, and B. Wiley. Freenet: A distributed anonymous information storage and /etrieval system. In Proceedings of the Workshop on Design Issues in Anonymity and Unobservability, Berkeley, California, June 2000.

[30] MojoNation web site - http://www.mojonation.net

[31] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, S.R. Gummadi, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: architecture for global-scale persistent storage. In Proceedings of ACM ASPLOS. ACM, November 2000.

[32] M. Khambatti, K. Ryu, and P. Dasgupta. Structuring peer-to-peer networks using interest-based communities. In Proceedings of the International Workshop On Databases, Information Systems and Peer-to-Peer Computing (P2PDBIS), Berlin, Germany, September 2003.

[33] P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In Proceedings of the Eighth Workshop on Hot Topics in Operating Systems, May 2001.

[34] M. Shaw and D. Garlan. Formulations and formalisms in software architecture. In Jan van Leeuwen, editor, Computer Science Today: Recent Trends and Developments, pages 307-323. Lecture Notes in Computer Science 1000. Springer Verlag, 1995.

[35] A.B. Stubblefield and D.S. Wallach. Dagster:censorship-resistant publishing without replication. Technical Report Technical Report TR01-380, Rice University, Dept. of Computer Science, July 2001.

[36] Y. Chen, R.H. Katz, and J.D. Kubiatowicz. Scan: A dynamic, scalable and efficient content distribution network. In Proceedings of International Conference on Pervasive Computing, 2000.

[37] R. Dingledine, M.J. Freedman, and D. Molnar. The FreeHaven project: Distributed anonymous storage service. In Workshop on Design Issues in Anonymity and Unobservability, pages 67-95, July 2000.

[38] Groove web site. http://www.groove.net

[39] S. Hand and T. Roscoe. Mnemosyne: Peer-to-peer steganographic storage. In Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), MIT Faculty Club, Cambridge, MA, USA, March 2002.

[40] M.A. Baker, and Matthew Grove, Tycho: A Wide-area Messaging Framework with an Integrated Virtual Registry, Special Issue on Grid Technology of the International Journal of Supercomputing, (eds) George A. Gravvanis, John P. Morrison and Geoffrey C. Fox, Springer, Volume 42, pp 83-106, ISSN: 1573-0484, March 23, 2007.

[41] GridFTP, http://www.globus.org/grid_software/data/gridftp.php

[42] NaradaBrokering, http://www.naradabrokering.org/

[43] Globus, http://globus.org

[44] IRC, http://en.wikipedia.org/wiki/IRC

[45] R-GMA, http://www.r-gma.org/

[46] Jini, http://www.jini.org/

[47] Apache AXIS, http://ws.apache.org/axis/

[48] VERA, http://vera.rdg.ac.uk

[49] Vindolanda, http://vindolanda.csad.ox.ac.uk/

[50] XDB, http://xdb.vera.rdg.ac.uk/

[51] Necho, http://acet.rdg.ac.uk/projects/necho/

[52] Sloan Digital Sky Survey, http://www.sdss.org/

[53] WGET, http://www.gnu.org/software/wget/

[54] Azureus, http://azureus.sourceforge.net/

[55] VOTechBroker, https://portals.rdg.ac.uk/votb

[56] M.A. Baker and G. Smith, GridRM: An Extensible Resource Monitoring System, the proceedings of IEEE International Conference on Cluster Computing (Cluster 2003), Hong Kong,

IEEE Computer Society Press, ISBN 0-7695-2066-9. 207-215, 2003,

[57] Self-Organizing ICT Resource Management (SORMA), http://www.iw.uni-karlsruhe.de/sorma.

[58] M.A. Baker and R. Boakes, Slogger: A Profiling and Analysis System based on Semantic Web Technologies, Special Issue of Scientific Programming on Large-Scale Programming Tools and Environments, (editors) Barbara Chapman and Dieter Kranzlmuller, International Journal of Scientific Programming, IOS Press, Vol. 16 (Number 2-3) 183-204, ISSN 1058-9244, 2008

[59] Map Server, http://mapserver.org/

[60] JSR-000168 Portlet Specification, http://jcp.org/aboutJava/communityprocess/review/jsr168.

[61] Gridsphere Portal Framework, http://www.gridsphere.org/gridsphere/gridsphere.

[62] G.M Smith, M.A. Baker and Javier Diaz Montes, A Web 2.0 User Interface for Wide-area Resource Monitoring, 15th Mardi Gras Conference, Baton Rouge, Louisiana, 30 January - 2 February 2008, , ISBN 978-1595930-835-0.

[63] RESN, http://acet.rdg.ac.uk/projects/resn/

[64] Sun SPOT, http://www.sunspotworld.com/

[65] RDF, http://www.w3.org/RDF

[66] SPARQL, http://www.w3.org/TR/rdf-sparql-query/