Service-oriented Middleware for Hybrid Environments

Andrew Harrison
School of Computer Science
Cardiff University
Cardiff, Wales, UK
a.b.harrison@cs.cf.ac.uk

Dr. Ian Taylor
School of Computer Science
Cardiff University
Cardiff, Wales, UK
i.j.taylor@cs.cf.ac.uk

ABSTRACT

This paper discusses the challenges facing hybrid, dynamic environments with regard to handling stateful resources and contextual interactions and describes strategies we have and are developing to enable applications to exist in these diverse environments. In particular we describe how combining Grid computing and Web services technologies with Peer-to-Peer technologies can be used to create flexible and dynamic Service Oriented Architectures that support stateful interactions in a loosely coupled way. We describe WSPeer, a SOAP based middleware that enables sophisticated message exchanges in Peer-to-Peer environments, and introduce WSKPeer, a micro-edition of WSPeer currently under development.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

1. INTRODUCTION

Grid computing has traditionally focussed on distributed data management and computation in environments connecting high-end resources with stable addresses across fast networks. In recent years however, we have seen the concept of the Grid expanding to include nodes that are resource constrained or inherently unreliable in terms of connectivity and addressability including nodes *at the edges of the Internet* [31], mobile devices and sensors. The motivation behind this expansion is the belief that both Grids and *nodes at the edges* can benefit from the extended possibilities offered by integration.

An example of a such a hybrid system that incorporates diverse entities, from centralized, powerful and static computers to mobile devices and laptops, down to sensors, is a weather modeling system such as proposed by the LEAD [11] project. In its normal running state, the system feeds data from the sensors in the field back to a static Grid infrastructure for monitoring. If the monitoring leads to the discovery of a possibly emerging significant event, such as a severe storm, a number of simulations to ascertain the likelihood of adverse conditions arising are spawned. Depending on these preliminary results, further simulations may be spawned at a finer level of granularity. If these detailed simulations suggest immanent adverse conditions, then events are propagated back into the field to a heterogeneous set of nodes such as civic centers, emergency vehicles, hospitals and doctors, to enable them to prepare for an emerging situation. The core characteristics this system must possess include:

- ability to traverse heterogeneous environments.
- ability to handle events.
- ability to dynamically expand and contract, i.e. discover new resources at runtime to perform processing based on incoming events, retrieve the results and make decisions based on these results.

We believe that Peer-to-Peer (P2P) technologies can make significant contributions towards the realization of such systems because they have been developed to handle heterogeneous and unstable environments. Synergies between Grid computing and P2P computing have been noted by a number of authors [32] [12] [25] and P2P technologies are perceived to be particularly useful to Grid computing with regard to scalable resource distribution and discovery [27] [21] [33] and CPU cycle scavenging [2]. Research into the integration of mobile devices in Grid environments is also ongoing [1] [19] which will enable users to manage computations from limited devices as well as provide aggregations of devices that can perform computations for the Grid. P2P technologies are also being proposed for these environments [20] [29] to enable local scalable decision making. The integration of sensors into Grids [14] [13] will extend computational Grids by introducing real-time data acquisition into the computational loop allowing systems to respond dynamically to data driven events, and decentralized architectures have been proposed in this context as well [34].

While the integration of these different environments opens up many new possibilities, it also introduces new challenges, in particular how to facilitate complex interactions that require the maintenance of state or context across numerous communications despite heterogeneity and unreliability of nodes. Further, the ability for middleware and services to span hybrid hosting environments is vital to address application scenarios that span heterogeneous networks of

Grids, whilst maintaining the transparency of the resources they wish to expose through common interfaces.

This paper discusses issues surrounding these challenges and describe strategies we are employing to address them. In particular we revisit the concept of Service Oriented Architecture (SOA) and show how the combination of Web and Grid services standards combined with P2P techniques for discovery and addressing can be used to develop SOAs that support complex interactions across heterogeneous environments. We discuss a framework called WSPeer [17] [16], which directly addresses some of these issues, and we introduce the micro-edition of WSPeer, called WSKPeer, which is a framework under development aimed at providing similar cross-environment service capabilities to WSPeer, but for highly constrained devices.

The following section describes the challenges facing SOA in hybrid environments, particularly the maintenance of stateful interactions. Section 3 describes WSPeer and Section 4 introduces WSKPeer and its three core modules. Section 5 describes planned future research and development and draws some conclusions from the issues discussed.

2. SERVICE ORIENTED ARCHITECTURE

Service Oriented Architecture (SOA) is perceived as an ideal paradigm to integrate heterogeneous environments because of the loose coupling encouraged by *services*. In the absence of a generally accepted definition of the term *service* we suggest the following properties are central to the loose coupling encouraged by the concept:

- A service is a view of some resource - a software asset, a business, a hard disk, anything that provides some capability. Implementation detail is hidden behind the service interface.
- Services communicate using messages. The structure of the message and the schema, or form, of its contents are defined by the interface.
- Services are stateless. By this we mean that all the information needed by a service to perform its function is contained in the messages used to communicate with it.

Services discover and communicate with each other using the *publish, find, bind* [15] [28] triad of operations. A service publishes its interface definition to the network, a service consumer finds the definition and, using the information in the definition, is able to bind (resolve the address and send messages), to the service. An important aspect of SOA is the *just-in-time integration of applications* [22] facilitated by these three operations. In other words the interface definition which describes the form of messaging combined with facilities for publishing and discovering it enables late-binding between entities to create dynamic aggregations of services.

The move towards SOA in Grid computing via the Open Grid Services Architecture (OGSA) is making interoperability with similar initiatives such as the Sensor Web Enablement (SWE) [6] initiative of the Open Geospacial Consortium (OGC) and the Open SensorWeb Architecture (OSWA) [5] more feasible. Although P2P systems have not embraced Web services technologies in such a cohesive manner as is represented by OSGA, OSWA and SWE, many architectures such as Gnutella [3], Freenet [10] and JXTA [23] are exponents of SOA:

- they follow the publish, find, bind pattern.
- they communicate using a message-oriented paradigm. All information required to process a communication is contained in the message.
- they perform just-in-time integration. In the case of Gnutella for example, a dynamically created request path represents an integration designed to fulfill the request.

However, traditionally P2P systems only support simple interactions such as file storage and retrieval.

2.1 Services vs Resources

Despite the huge difference between Grid computing resources and those that make up mobile and sensor networks, both types of environment can be seen as resource-centric. In Grid computing hardware resource characteristics are important because the properties of a particular machine for example - system configuration, memory capacity, load and network link speed - play an important part in deciding whether it can or should be used as part of a particular computation. Furthermore resources such as jobs and files may need to be tracked through time. Likewise in mobile and sensor networks, because of resource constraints, connectivity issues and location-awareness, resource characteristics are tightly bound to service capability. In the case of sensors the resource can often be viewed as almost analogous to the service it provides. Furthermore hybrid systems such as the weather forecast system described in Section 1 require sophisticated messages which encapsulate, not just data, but the context in which this data is generated or computed. This is essential for a number of reasons: The system must be able to assess the trustworthiness of, not just other nodes in the system, but the data generated by these other nodes. Likewise events that encapsulate state need to be understood in the context of a continuum.

The modeling of state in service-oriented Grid computing is an ongoing challenge. The Web Services Resource Framework (WS-RF) [24] represents the latest incarnation of OGSA. It comprises a suite of specifications that encompass managing stateful resources, groups of services and subscriptions to and notifications of topics. All these capabilities are highly relevant to the kind of hybrid environments under discussion. At the center of the specifications is the concept of a WS-Resource. A WS-Resource is some entity with state which is accessed through a service interface. WS-Resources are used to represent, not just entities such as files and jobs, but subscriptions to topics and group membership tokens. However, a WS-Resource is modeled as the combination of a resource identifier and a service endpoint. The service endpoint represents the service that will be able to de-reference the resource id to some actual resource. This coupling encourages the WS-Resource to be used as a global pointer to some entity that “resides” at the given endpoint [26]. This is a false presumption particularly in dynamic environments because services and resources may change address or physically move thus breaking the relationship between the two elements and invalidating the WS-Resource.

We suggest that the strict separation of services from resources is required to ensure scalability and robustness in dynamic, hybrid environments:

- heterogeneity is more easily masked because the service interface is divorced from the underlying host or back end resource.
- resource unreliability (failure, replacement, mobility, address inconsistency) can more easily be accommodated because a service can adopt a different resource or a resource can adopt another service.
- A proliferation of global pointers to entities with state produces fragile systems [37], particularly systems characterized by high churn and mobility. The separation of instances of resources from exposed interfaces allows multiple resources to be addressed by a single service which may be an interface to a sub domain or group of resources.

In the following section we describe the Web services framework WSPeer and the strategies used by it to employ the complex and stateful interactions supported by WS-RF in hybrid environments while still maintaining a separation between services and resources.

3. WSPEER

While Web services are considered an ideal means for building SOA, most implementations are built for enterprise scale application servers. Furthermore, support for more complex message exchange patterns such as those that model state or enable publish/subscribe relationships are rarely supported in light-weight implementations.

WSPeer began as a Web services API for the Triana problem solving environment [35] to allow the creation, publishing, discovery, invocation and composition of Web services for its workflows. Triana uses a P2P-oriented application level API called the GAP to abstract the implementation details of service-oriented middleware allowing arbitrary protocols to be implemented as GAP bindings as long as they can fulfill the basic operations of publish, find and bind. WSPeer constitutes the GAP Web services binding and therefore is designed as a high level, P2P oriented API to SOA, and in particular, Web services technologies. The high-level API insulates an application from underlying protocol details while the P2P orientation means WSPeer treats service provider and consumer roles symmetrically without the need for a service container at the server side. Instead, an application adds itself as a listener to events fired by the system, for example, when a service is created, published, discovered or when a message (service request or response) arrives.

Apart from the usual HTTP transfer protocol binding, WSPeer supports JXTA, Peer-to-Peer Simplified (P2PS) [36] and the Styx protocol. P2PS is a domain independent framework for developing P2P applications. It is similar to JXTA in that it uses the pipe abstraction to define communication channels and supports rendezvous peers for advertisement caching and the creation of peer groups. An important aspect of the use of JXTA and P2PS in WSPeer, is to combine the strengths of P2P systems, namely decentralized publishing and discovery, with the strengths of Web services technologies, namely the description of sophisticated service interfaces and message exchanges. Furthermore both JXTA and P2PS use logical addressing to identify endpoints. These logical addresses are resolved at runtime to network addresses allowing for location transparency of nodes. The WS specifications supported by the WSPeer include WS-Security, WS-Addressing and WS-Transfer and WS-RF (including WS-Notification).

In employing WS-RF and WS-Notification over P2P topologies, we have employed the logical addressing capabilities of JXTA and P2PS [18]. This allows the tight coupling between resource identifier and a service endpoint apparent in the WS-Resource construct to be de-coupled. The JXTA addressing mechanism uses a unique peer id as the basis for a logical address. P2PS uses two mechanisms. The first is similar to JXTA, the second, still under development, uses hierarchies of *concepts* to point to peers and peer groups. The form of this naming scheme is a simple hierarchical URL. Hierarchies represent traversals of the P2P network via peers or groups associated with a given concept. This mechanism is similar to naming schemes such as the Intentional Naming Scheme [8] and the semantic ontology-based scheme described in the context of a P2P hypercube topology [30]. Concepts are not uniquely mapped to endpoints as is the case with the peer id mechanism. This allows addresses to be resolved to different peers at different times which is useful in the context of WS-Resources as it enables mobility of both service and resource. Furthermore, addresses may resolve to peer groups allowing local resolution of the resource identifier in the WS-Resource. This alleviates the problem of WS-Resources being interpreted as global pointers.

The combination of a number of elements brought together through WSPeer addresses the properties defined in Section 2 as characteristic of hybrid environments. The high level API and P2P architecture allows an application to seamlessly traverse heterogeneous environments using SOAP as a common language irrespective of the role (provider, consumer) of the application in any particular message exchange. Furthermore the integration of the support for Web services specifications that can describe complex interchanges, including state management and event notification, with P2P mechanisms of discovery and late binding through logical addressing maintain the separation of services and resources and therefore open up the possibility for applications to manage unreliability as well as the dynamic event-driven expansion and contraction of the system.

4. WSKPEER

WSPeer is designed for environments that are not resource constrained and is therefore not suitable for restricted devices. The libraries required by WSPeer, including the Java Runtime Environment itself, are simply too large. For example WSPeer uses Apache's Axis 1.5 [9] which in turn makes use of XML libraries such as the Apache Xerces XML parser which is itself a megabyte in size. As a result we have been developing a package called WSKPeer that provides the same high level interface to core SOA capabilities as WSPeer, but for constrained environments.

WSKPeer builds on the lessons learned in developing WSPeer and uses only a subset of core Java classes and a minimal set of libraries - essentially the kXML [4] library which includes an XML pull parser and some core DOM classes. From these simple data structures we have been building a Web services framework that can support service description parsing, including complex types and sophisticated message exchanges that support notions of state and events, in particular WS-RF and related specifications, despite the limited availability of processing power.

The core of WSKPeer is a simple transport and transfer independent SOAP message processing API which can be plugged into any back-end application. To receive messages, the application must add a listener component. Message generation can make use of other WSKPeer components such as service description and XML schema parsing, or can use 'hardcoded' message types if the application knows these messages at compile time or is to be deployed into a very restricted environment. Likewise capabilities such as discovery and publishing are optional depending on the circumstance. Because WSKPeer is compatible with J2ME it can be used with other J2ME classes such as the MIDP GUI classes, but can also be used in situations where SOAP processing needs to be lightweight and not reliant on the hosting environment, for example mobile agents or distributed executables on a Grid. The following sub sections describe the core WSKPeer modules.

4.1 Messaging API

WSKPeer aims at message-orientation, a cornerstone of SOA, and hence the Message abstraction is a top-level citizen in the framework. Figure 1 shows the messaging WSKPeer API. It is made up of four core classes:

- **Message** This class represents the actual SOAP message. It contains a *MessageEnvelope* which in turn contains kXML DOM *Node*s defining the body and header elements of the envelope. A *Message* can also have metadata assigned to it via properties.

- **MessageListener** This interface is implemented by components wishing to receive arriving *Message*s. It defines a single *receive(Message)* method.

- **Port** This interface represents a component that is capable of optionally sending messages (*send(Message)*), sending and receiving messages synchronously (*exchange(Message):Message*) and adding *MessageListeners* (*addMessageListener(MessageListener)*). When a message arrives at a port, it notifies the registered listeners.

- **Context** This interface performs some processing on a message, either on its outward (*processOut(Message)*) or inward (*processIn(Message)*) journey. This may constitute processing WS-Addressing, WS-Security or WS-RF information as examples. A *Context* should sit between the *MessageListeners* and the *Port* in the message flow.

A *Pipeline* class is also defined. This is a composite of the *Port* and *MessageListener* interfaces enabling the chaining of contexts for compound and ordered message processing. An application can create a *Pipeline*, give it a *Port*, add itself as a listener to the *Pipeline* and construct a series of contextual processes that the *Pipeline* should invoke.

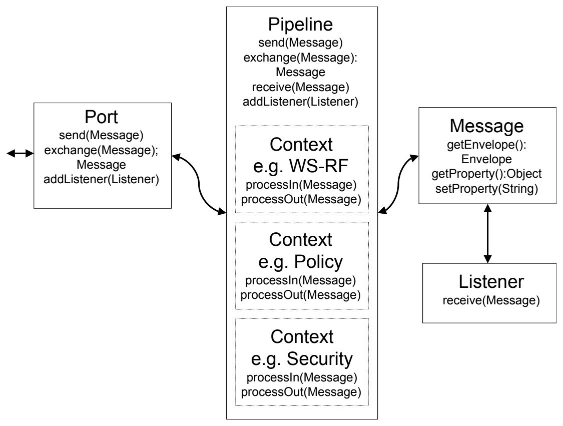

Figure 1: WSKPeer messaging API.

4.2 Data Binding

Message-orientation frees us from the trappings and overhead that are often associated with the RPC model. Without the notion of a remote procedure being called we can restrict ourselves to a document-centric view; that is, the message being sent or received is simply an XML document. Therefore, we presume a document-literal SOAP encoding style. Experience in interoperability as well as the direction of emerging WS-* standards suggests this encoding style will become most widely used. Furthermore the generation of stubs, i.e. local objects that represent instances of remote objects, is not required as we are not conceptually invoking a remote object.

Likewise the process of data binding usually involves generating and compiling code locally that models the schemas defined for complex types. This generation requires a certain level of computational power as well as the memory to store the libraries and compiler to perform the generation which may not be accessible to constrained devices. Furthermore, stub generation, firstly, usually presumes a relatively static environment in which well known types or services are reused again and again for the lifetime of an application. This cannot be presumed in a dynamic environment. Secondly the conversion of an XML schema to a programming language defined object usually implies the presence of a programmer for whom handling a programming language object is easier than handling an XML document. Again, in an environment made up of mobile devices and sensors, human intervention cannot be presumed. Therefore, as well as not using stubs to represent remote services, we have designed an infrastructure for parsing and accessing complex types without the need for local Java classes or the introspection of them. In fact, in WSPeer, we have found that generating local Java objects does not add much functionality to dynamic message exchanges. Although WSPeer has optimized this generation by creating Java classes directly as an array of bytes, thus circumventing compilation, at runtime these classes still need to be introspected (using the Java reflection mechanism) to ascertain their capabilities. Therefore one is limited to the reflection API. In WSKPeer we use a generic *Type* class that provides its own reflection mechanism for discovering and setting the content fields of the type, as well as generating an XML serialization of the type for inclusion in a message. The reflection mechanism is analogous to the Java reflection API, however we do not need to generate or compile Java classes to achieve the same result.

As part of its modular architecture, WSKPeer provides a *MessageModule* interface. These components are capable of creating families of *Messages*, for example WS-Notification related messages. This approach alleviates some processing and coding complexity if an application knows it will be handling certain types of messages because *MessageModule* instances hardcode some common elements of messages. The interface defines a single method: *public Message createMessage(String name, Object[] params)* which allows an application to insert specific data into a message template. This approach has been used to implement support for WS-RF and WS-Notification message exchanges.

4.3 Service Description API

To achieve optimized service description generation, WSKPeer defines a generic service and message description interface that captures a subset of properties of a description. This means the stored description and the actual description are not necessarily isomorphic as is usually the case. This approach is also taken in WSPeer, but for WSKPeer the interface has been honed down. The decision to create a simplified and generic interface to service description is partly based on the belief that although WSDL is currently the de facto language for describing Web services, other emerging technologies, for example the SOAP Service Description Language (SSDL) [7], are likely to play a role in service description in the future, and partly because certain protocols, in particular those designed for mobile devices and sensors define their own description and discovery mechanisms which may be more suitable for applications to interpret. Therefore the application is shielded from these protocol specific features through a high level interface. So for example, when parsing WSDL we make the presumption that the binding is a SOAP binding and its encoding style is document-literal thus eliminating the requirement to store this data. However the transport is of course not presumed as we want to make WSKPeer available to any number of transport and transfer protocols, and is stored as an identifier in the service description so that a suitable *Port* implementation can be used to send messages at a later date.

This description interface is still work in progress as it has not been applied to any other description technologies beyond WSDL. We intend to develop support for SSDL as the next step.

The API contains three components. The *ServiceDescription* has a location (where the description itself was retrieved from) and provides a service endpoint, a target namespace and a transport name. This component has one or more *MessageExchanges*. A *MessageExchange* encapsulates a series of messages that comprise a complete exchange. Each exchange is uniquely identified by a qualified name (QName) and a type of exchange. A *MessageExchange* in turn maintains one or more *MessageTypes*. *MessageTypes* likewise have a QName as well as a QName representing the XML element that it is constructed from. This can be cross referenced against the types generated from XML schema to retrieve an actual XML element instance. A *MessageType* also has a field representing how the message is used, for example whether it is an incoming or out going message or a fault. Finally it can have an action associated with it. This defines some semantics of how the message should be interpreted. Currently this is used to house a WS-Addressing Action URI which is a common way of defining message semantics in a number of specifications.

5. DISCUSSION AND CONCLUSIONS

A general issue with regard to Web services technologies and limited devices is of course the verbosity and processing requirements of XML. XML optimization is an area of active research and we will be exploring avenues for developing support for it. However there is currently no agreed upon standard. We foresee the use of XML in constrained environments being restricted to particular types of communication. A potential usage for WSKPeer is in defining standards-based negotiation and control layers. As an example, [29] describes a scenario in the field of mobile paramedical emergency operations in which mobile medical devices acquiring patient data in real-time communicate with ambulance access points. Before data can be streamed to these access points however, protocols need to be established during a negotiation stage in which participants can be authenticated, stream endpoints and binary formats defined, and stream control and monitoring protocols established. It is at this stage that standards based messaging supporting complex exchanges is particularly useful as it represents a common means of defining diverse interactions and optimizations. Likewise systems that are event based, such as the scenario discussed in Section 1, benefit from standard means of subscription and notification because events can travel freely through the system despite the heterogeneous nature of the participants. The support for WS-Notification by constrained devices in such environments represents another potential role for WSKPeer.

There are still a number of issues to address in WSKPeer. Primary focus will now be on developing the P2P bindings - a JXTA J2ME binding as well as P2PS for J2ME. These will form the basis for defining the publish and discovery APIs and address the separation of service interfaces from stateful resources (see Section 2.1). A Bluetooth binding is also currently under development. The core APIs described in section 4 have been developed and, along with an HTTP Port implementation, a WS-RF compatible WSDL parser and required libraries, bundle into a jar file of approximately 65kb.

In this paper we have described how the requirement for complex and stateful interactions can be supported in heterogeneous environments through a combination of Web and Grid service messaging standards and P2P technologies of discovery and addressing. This combination provides a mechanism for developing flexible and dynamic SOAs. We discussed WSPeer in the context of this combination and introduced WSKPeer - a micro-edition of WSPeer still in development.

While we do not suggest that even the smallest, most limited devices should be able to process complex XML message interactions, we do believe it is possible to push the responsibility for generating and consuming interoperable messages to the edges of the network. The deeper the shared messaging languages penetrate the environment, the more flexible, fault tolerant and inclusive the network can be.

6. REFERENCES

[1] AKOGRIMO Integrated Project. See `http://www.akogrimo.org`.

[2] BOINC - Berkeley Open Infrastructure for Network Computing. `http://boinc.berkeley.edu/`.

[3] Gnutella. `http://www.gnutella.com/`.

[4] kXML. `http://kxml.sourceforge.net/`.

[5] Open SensorWeb Architecture. `http://gridbus.cs.mu.oz.au/sensorweb/`.

[6] Sensor Web Enablement Working Group. `http://www.opengeospatial.org/projects/groups/sensorweb`.

[7] SOAP Service Description Language (SSDL). `http://www.ssdl.org/`.

[8] E. S. W. Adjie-Winoto and H. Balakrishnan. An Architecture for Intentional Name Resolution and Application-level Routing. Technical Report MIT/LCS/TR-775, MIT, 1999.

[9] Apache Project. Apache Web Services Project - Axis, July 2005. `http://ws.apache.org/axis/`.

[10] I. Clarke, S. G. Miller, O. Sandberg, B. Wiley, and T. W. Hong. Protecting free expression online with freenet. *IEEE Internet Computing*, pages 40–49, January, February 2002.

[11] K. Droegemeier et al. Service-oriented environments in research and education for dynamically interacting with mesoscale weather. *Computing in Science and Engineering*, 7(6):12–29, November/December 2005.

[12] I. Foster and A. Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *Proceedings of the* 2^{nd} *International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[13] G. Fox et al. Management of Real-Time Streaming Data Grid Services. In *Invited talk at Fourth International Conference on Grid and Cooperative Computing (GCC2005), Bejing, China*, 2005.

[14] M. Gaynor et al. Integrating wireless sensor networks with the grid. *Internet Computing, IEEE*, 8(4):32–39, July-Aug 2004.

[15] D. Gisolfi. Web services architect: Part 1. `http://www-128.ibm.com/developerworks/webservices/library/ws-arc1/`, April 2001.

[16] A. Harrison and I. Taylor. Dynamic Web Service Deployment Using WSPeer. In *Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, pages 11–16. Louisiana State University, February 2005.

[17] A. Harrison and I. Taylor. WSPeer - An Interface to Web Service Hosting and Invocation. In *HIPS Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models*, 2005.

[18] A. Harrison and I. Taylor. The Web Services Resource Framework In A Peer-To-Peer Context. *Accepted for publication in Journal of Grid Computing*, 2006.

[19] M. Humphrey and D. C. Chu. Mobile OGSI.NET: Grid Computing on Mobile Devices. In *Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, 2004.

[20] J. Hwang and P. Aravamudham. Middleware Services for P2P Computing in Wireless Grid Networks. *IEEE Internet Computing*, 8(4):40–46, 2004.

[21] A. Iamnitchi, I. Foster, and D. C. Nurmi. A Peer-to-Peer Approach to Resource Location in Grid Environments. In *11th Symposium on High Performance Distributed Computing (HPDC 11)*, 2002.

[22] IBM Services Architecture Team. Web Services architecture overview. `http://www-128.ibm.com/developerworks/webservices/library/w-ovr/`, September 2000.

[23] Project JXTA, July 2005. `http://www.jxta.org`.

[24] Karl Czajkowski et al. The WS-Resource Framework, March 2004. `http://www.globus.org/wsrf/`.

[25] J. Ledlie, S. J., S. M., and J. Huth. Scooped, Again. In *Peer-to-Peer Systems II: Second International Workshop, IPTPS 2003*, 2003.

[26] M. Little, J. Webber, and S. Parastatidis. Stateful interactions in Web Services: a comparison of WS-Context and WS-Resource Framework. Web Services Journal, May 2004.

[27] C. Mastroianni, D. Talia, and O. Verta. A Super-Peer Model for Building Resource Discovery Services in Grids: Design and Simulation Analysis. In *European Grid Conference EGC 2005, Amsterdam, The Netherlands*, 2005.

[28] M. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *Fourth International Conference on Web Information Systems Engineering (WISE '03).*, 2003.

[29] A. Riposan et al. Mobile Peer-To-Grid Architecture for Paramedical Emergency Operations. In *HealthGrid 2006*, 2006.

[30] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl. A Scalable and Ontology-Based P2P Infrastructure for Semantic Web Services. In *Second IEEE International Conference on Peer-to-Peer Computing (P2P2002)*, 2002.

[31] C. Shirky. Modern P2P Definition. `http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html`, 2000.

[32] D. Talia and P. Trunfio. Toward a synergy between P2P and grids. *Internet Computing, IEEE*, 7(4), 2003.

[33] D. Talia and P. Trunfio. Web Services for Peer-to-Peer Resource Discovery on the Grid. In *DELOS Workshop: Digital Library Architectures*, 2004.

[34] C. Tham and R. Buyya. SensorGrid: Integrating Sensor Networks and Grid Computing. *CSI Communications, Special Issue on Grid Computing, Computer Society of India*, July 2005.

[35] The Triana Project. See web site at: `http://www.trianacode.org`.

[36] I. Wang. P2PS (Peer-to-Peer Simplified). In *Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, pages 54–59. Louisiana State University, February 2005.

[37] J. Webber and S. Parastatidis. Horses for Courses: Services, Objects, and Loose Coupling - Integration without compromise. `http://www2.sys-con.com/ITSG/virtualcd/WebServices/archives/0401/webber/index.html`, 2004.