Abstract—This paper introduces WSPeer, a high level interface to hosting and invoking Web services. WSPeer aims to support the diversification of Web service deployments by providing a pluggable architecture that can be used to host Web services in a number of different ways. In particular, we are interested in the cross-fertilisation between Web services and Peer-to-Peer (P2P) systems by enabling entities to act as both service providers and consumers in a flexible and extensible way. Further, we support the behaviour needed within a P2P environment by allowing Web services to be dynamically deployed through the use of a lightweight container that is more suited to highly transient connectivity. This approach allows standard Web services to be deployed within a P2P environment and yet still adhere to a Service Oriented Architecture. We argue therefore that both Web services and P2P are exponents of Service Oriented Architecture, approaching the same goals from very different directions, and that a combination of the strengths of these approaches can lead to more robust and ubiquitous Web service and P2P deployments.

I. INTRODUCTION

This paper introduces WSPeer, an Application Programming Interface (API) for hosting and invoking Web services. The motivation behind WSPeer rests on the belief that the hosting, discovery and invocation mechanisms of Web services need to diversify. Hence we use the term ‘Web services’ here in the widest possible sense without reference to certain mechanisms used in current Web service deployments such as transport and discovery protocols and hosting environments.

We argue that Peer-to-Peer (P2P) systems represent an important branch in the development of Service Oriented Architectures (SOA) and describe how the issues confronted and lessons learnt by these systems are being integrated into and combined with more standard Web service technologies. If this diversification is to take place however, then service providers and requestors need to be protected from the ensuing complexity in order to make service provision and consumption manageable as well as ensuring adaptability in the face of new developments.

WSPeer is an attempt to construct an environment that enables diversity on the one hand while protecting from it on the other. Section II gives a brief description of current implementations of Web services and P2P systems showing how they are influencing each other. Section III gives an overview of the WSPeer architecture, describing how it is capable coping with diverse implementations. Section IV briefly describes the two current implementations of WSPeer.

II. BACKGROUND

The term ‘Service Oriented Architecture’ is widely used, yet interpretations of its meaning vary according to who uses it and what the perceived benefits of an SOA are. Furthermore Web services are currently considered the most suitable technology for developing SOAs and as a result it can be difficult to disentangle Web services implementation detail (for example the use of XML and Universal Discovery, Description and Integration (UDDI) [1]) from a more abstract overview. A very broad definition of the intent of an SOA is given by [2]:

SOA is an architectural style whose goal is to achieve loose coupling among interacting software agents.

In an SOA this challenge of loose coupling is addressed through the notion of a ‘service’. Most commentators [2] [3] [4] [5] are agreed on three primary attributes that characterise a service:

1) A service is a logical view of a software asset. Its purpose is to hide the implementation detail of the underlying software.
2) A service has a well-defined public interface. By ‘well-defined’ we mean the interface describes a discrete entity with a certain number of operations and a point of contact.
3) Communication with a service takes place using a message-based communication model defined by the interface.

A further quality of the messaging format required by certain authors is that it must also be platform neutral [4] [3].

An SOA builds on the concept of a service. It is a collection of services capable of interacting in three ways, commonly referred to as ‘publish, find and bind’. In other words a service must be able to make its interface available to other services (publish), other services must be capable of discovering the interface (find), and finally services must be able to connect to one another to exchange messages (bind).

The loose coupling of an SOA is achieved firstly through the separation of data exchange from the software agents involved in the exchange, and secondly through the discrete
nature of the services. Some commentators additionally stress the compositional nature of an SOA [6] [5]. Because services are discrete, well-defined entities they can be used as building blocks in creating larger processes and workflows. This modularity enhances the loose-coupling of the architecture. Although the benefits of an SOA are numerous, including code reuse (new processes can be created by composing existing services) and code wrapping (legacy applications can be wrapped in a compliant interface), the main goal, we believe, of an SOA is to allow heterogenous nodes to interact. Heterogeneity is the one obstacle to ubiquity and hence all SOA betray an interest in achieving ubiquity.

Although, Web services are perceived to be the ideal mechanism for developing an SOA, other systems also display suitable criteria. For example, Jxta [7] allows services to be published, located and dynamically bound to within a P2P environment. From this perspective, there is little conceptual difference between Web services and Jxta except that Jxta exploits decentralised techniques to publish and search for the service capabilities across multi-hop layers of interconnected peers and/or rendezvous peers. Likewise both systems support service interface description. Whereas Web services use Web Service Definition Language (WSDL) [8] to represent a service interface, Jxta uses a Jxta service descriptor. These similarities have prompted efforts to integrate Web service technologies such as Simple Object Access Protocol (SOAP) [9] and WSDL into Jxta networks [10].

Further, P2P systems with a more limited scope, such as Gnutella [11] and Napster [12], exhibit the same ability to publish, find and bind (in these cases to media files) within their architectures, with Napster bearing a resemblance to the use of UDDI within the Web services community. Gnutella, on the other hand employs in-network discovery mechanisms which can be used to form impromptu network mechanisms between peers in order to search for content. In the Web service world, this would be analogous to searching for WSDL files.

The peculiarities of the requirements of the Web services and P2P communities have resulted in both strengths and weaknesses in the two approaches. The reliance on organisational infrastructure in Web and Grid service networks has lead to rather centralised client/server based mechanisms of communication and discovery, specifically the use of HTTP (or extensions to it) as a transport protocol and UDDI as a discovery mechanism. The client/server nature of these networks potentially inhibits their scalability because the number of server entities does not grow proportionately with the overall number of nodes. This creates communication bottlenecks and increases the stress on the servers and hence the likelihood of their failure.

However this organisational approach has also lead to the development of open standards that are shared between interested parties and address the need for complex inter-organisational communication. The two most important of these standards are SOAP and WSDL. These XML specifications allow not only interoperability and loose-coupling between service consumer and service provider, but also the ability to publish arbitrary services to the network, because service consumers can discriminate between them.

Current P2P implementations have not achieved this level of standardisation of message exchange or service description. Most P2P systems, Napster, Gnutella and Freenet [13] for example are limited to a single activity across the network - in the case of the first two, it is file sharing, in the case of Freenet, it is information storage and retrieval. Furthermore, two systems whose activities intersect cannot communicate because the message formats are proprietary to the individual system. On the other hand P2P systems have developed sophisticated mechanisms for dealing with discovery and the unreliability of nodes. This has lead to the development of networks that are scalable and robust in the face of node failure.

The suggestion that these two approaches - P2P and Web/Grid services - should, and indeed will, converge has already been made [14]. This assertion is supported by a number of recent developments. The latest version (11 February 2004) of the W3C Working Group’s Web Services Architecture(WSA) [4] takes into consideration P2P style message exchange and discovery. Research into P2P solutions are being applied to Web service discovery [15] [16] [17] and Web services are being harnessed for P2P style resource sharing [18]. On the P2P front, Jxta represents an attempt to specify a meta level for P2P applications potentially allowing interoperability between applications and now includes a SOAP binding [10] enabling standards-based message exchange and service description between Jxta peers.

The term ‘convergence’ however is perhaps misleading as it implies the loss of heterogeneity. The various communities and their specific requirements will not go away and hence neither will the heterogeneity they impose. ‘Cross-fertilisation’ is a more accurate term as it implies not only the continuation of existing disparate needs, but the evolution of new network topologies and technologies to support them. This is already happening. Web service description and query matching is an area of active research, including efforts to extend description capabilities using Semantic Web technologies in both client/server and P2P topologies [19] [20] [21]. These efforts signal that current standards and specifications are merely snapshots in the evolution of flexible and rich messaging technologies.

These developments convince us that Web/Grid/P2P (WGP?) services require flexible deployment and invocation options that protect underlying implementations from this flux. Furthermore we believe that as a consequence of these developments, Web services will be progressively perceived in terms of the core XML specifications such as SOAP and WSDL which lie at the centre of the Web service stack. The transport and discovery layers will be seen as incidental to the environment the Web service is deployed into or even incidental to particular invocations or search requests.

To address these issues we have been developing a system called WSpeer that attempts to achieve this protection and
flexibility by positioning itself between an application that is exposing itself - or parts of itself - as a service, and other services in the network.

III. WSPeer Overview

WSPeer acts as an interface to hosting and invoking Web services. It allows application code to work with potentially varying and evolving service architectures while maintaining a consistent interface. Figure 1 shows how WSPeer sits between application code and remote services acting as both buffer and interpreter.

WSPeer has three main aims:

1) To act as a complete interface to both publishing and invoking services, enabling applications to expose themselves as service oriented peers.
2) To be applicable to a variety of network architectures including ‘standard’ Web service architectures using technologies such as UDDI and HTTP, and P2P style networks.
3) To be flexible and extensible, allowing users to develop application specific service capabilities of their choice.

To achieve the first aim, WSPeer combines the strengths of the Web service and P2P approaches. WSPeer uses SOAP as it’s messaging format (via Apache’s Axis [22] SOAP engine) and WSDL for service description. In order to behave as a peer however, WSPeer breaks with some Web service traditions:

1) Although WSPeer allows synchronous discovery and invocation, it is essentially an asynchronous, event driven system in which components subscribe to events and are notified when and if responses are returned from remote services. Asynchronicity allows for P2P style interactions with unreliable nodes. Message oriented systems are usually asynchronous. However current Web service implementations are often synchronous due in part to the use of HTTP which maintains an open connection for return messages but also because Web services are often perceived to be the latest incarnation of distributed object technologies such as Corba, DCOM and RMI rather than message processing entities [23].
2) WSPeer does not require a container as is usual for entities hosting Web services. In the traditional scenario, a user deploys a module into a container and the container manages the requests for, and responses from, the deployed module. The user makes no attempt to intercept these processes. This model is sufficient for situations where the deployed module is an autonomous entity which performs its own processing or simply queries a back-end with a static location. However, the scenario becomes cumbersome and un-intuitive if the user wishes to deploy an application which already has an established environment or requires user input at runtime. WSPeer reverses the power relationship between the deployed component and the environment used for deploying and exposing it, in effect allowing the component to become its own container. By adding itself as a listener to the WSPeer event structure an application is notified of all requests and responses either side of being processed by the underlying messaging system. WSPeer is added as an interface to the application rather than the application placing itself into an external entity. In the traditional scenario the container controls the service invocation environment. In WSPeer this environment is controlled by the application.
3) WSPeer takes this notion of reversing control further by allowing an application to generate and deploy a service which acts as an interface to a stateful object within the application. This essentially gives the service any state the instance it represents has. In fact each operation given to the service can map to a different stateful object in memory, allowing a service to be an interface to multiple objects. This facility is not designed to be a general pattern for modeling state in Web services such as file systems or databases, but rather a mechanism for exposing an arbitrary object as a standards compliant service/peer.

The second two aims are achieved through a pluggable architecture, allowing the insertion of different elements with differing capabilities. By plugging in different components, WSPeer can communicate with different entities without the application changing. For example, to locate a service in a UDDI registry, one would use a UDDI conversant component. But if one wanted to search for a service in a P2P network, a different component would be plugged in. The application code deals with WSPeer data structures, not those that are transmitted over the wire, so the application does not have to care where or how the service has been located, or what its definition looks like.

WSPeer is constructed as a tree of interfaces. Figure 2 shows this structure. The Server interface represents the server-side of the peer and the Client interface represents the client-side. On the server side, deploying a service involves taking a code source, generating a service interface description from it.
(WSDL for example), and creating an addressable endpoint which can be used to connect to the source. Publishing the service involves making the service endpoint and/or its interface description available to the network in some way.

On the client side, locating a service involves retrieving the endpoint of the service and possibly its interface description as well. Invoking involves passing a message to the remote endpoint and possibly awaiting a response. Nodes in the tree create implementations of their child nodes, register themselves as listeners to them, and receive notification of events fired by them. For example a Client will create an instance of a ServiceLocator and add itself as a listener to it. To initiate service discovery, the Client constructs a ServiceQuery object and passes this to its ServiceLocator. A ServiceQuery is an abstraction used by WSPeer to allow for varying kinds of query. The simplest ServiceQuery queries on the name of a service. More complex queries could be constructed from languages such as DAML [24]. When the ServiceLocator discovers a service it notifies the Client.

The aim of the tree structure and the notification system is to keep elements within the tree loosely coupled. This allows individual nodes in the tree to be replaced either at runtime or as part of a new implementation without disrupting the overall structure. To increase flexibility, implementations of child nodes can be registered with parent nodes as well, rather than being created by the parent node, allowing users to insert variations into the tree at any level.

All events are propagated upwards to the root of the interface tree - the Peer interface. Application code implementing the PeerMessageListener interface adds itself as an event listener to the Peer. Each of the interfaces below the Peer fire an event as the result of its activities and these events are brought together by the PeerMessageListener interface:

```java
public interface PeerMessageListener {
    public void peerMessageReceived(
        DiscoveryMessageEvent event);
    public void peerMessageReceived(
        PublishMessageEvent event);
    public void peerMessageReceived(
        ClientMessageEvent event);
    public void peerMessageReceived(
        ServerMessageEvent event);
    public void peerMessageReceived(
        DeploymentMessageEvent event);
}
```

IV. CURRENT IMPLEMENTATIONS

WSPeer currently has two implementations. The first uses 'standard' Web service technologies. The second uses a P2P framework called Peer-to-Peer Simplified (P2PS) [25]. P2PS came into being as a response to the complexity and overhead associated with Jxta. As its name suggests, it is a simple yet generic API for developing P2P systems. We chose to implement a proof of concept P2P binding for WSPeer using P2PS for a number of reasons:

1) The P2PS search mechanism can be extended to support attribute-based search, as opposed to the key-based search employed by Distributed Hash Table (DHT) systems such as [26] and [13]. DHT systems are really means of retrieving data rather than searching for data.

2) P2PS already uses XML to expose peers and services to the network. This makes extending this mechanism to incorporate Web service protocols a straight-forward task.

3) A Network Simulator (NS2) [27] agent [28] has been developed which uses an adaptation of P2PS as the interface to the simulator. This development will allow us to simulate large networks of peers publishing, discovering and invoking Web services in a distributed topology.

Although these two implementations interact with very different networks, from the application level, the processes involved in hosting and invoking services remain unchanged. This is shown in figures 3 and 4.

It is also worth noting that these implementations need not remain self-contained. A P2PS Client could use the UDDI enabled ServiceLocator defined in the standard implementation to search for services. Likewise, a P2PS Server could use the UDDI conversant ServicePublisher. We look briefly at the two implementations in turn. In particular
we concentrate on mechanisms in the P2PS implementation that address issues specific to enabling Web service standards to be applied to a P2P scenario.

A. HTTP/UDDI Implementation

The standard implementation is able to make the most of existing tools because of the common Web service technologies it uses. On the client side, the ServiceLocator implementation searches user defined UDDI registries for services. The client passes the locator a UDDIServiceQuery object to perform the match. This implementation of the ServiceQuery understands UDDI specific categories to search within. The default transport for an Invocation created in this implementation is HTTP, although HTTPSG (the transport used by Globus [29] for authenticated communication) is also supported. Here we make use of the facilities in Apache’s Axis [22] SOAP engine for generating stubs and invoking the remote endpoint (WSPeer actually extends the stub generation capabilities of Axis by generating stubs directly to bytes, bypassing source generation and compilation).

The server side is less orthodox. Here the Server implementation launches an HTTP server that listens for requests. This is similar to usual Web service implementations except for the absence of a container. Here the HTTP server is only launched once the application has deployed a service. The server’s capabilities are limited to listing available services and notifying the Server of incoming requests. If a request arrives then the Server gives the listening application a chance to handle the request directly. For applications that do not wish to deal with server-side message processing, this option can be turned off, in which case the Server invokes the underlying messaging engine (in this case Axis) directly. The ServicePublisher mirrors the client-side ServiceLocator in that it publishes services to UDDI registries. Figure 3 shows the four main processes of WSPeer using these standard technologies.

B. P2PS Implementation

A full description of Peer-to-Peer Simplified (P2PS) is beyond the scope of this paper. See [25] for a more detailed discussion. A brief explanation is required however, in order to understand how we apply its mechanisms. For the purposes of this discussion we focus on two main characteristics of P2PS.

1) P2PS peers use abstract communication channels, called pipes. This level of abstraction is used because peers are identified by a logical id, not physical address. Identifiers allow for multiple transport protocols to co-exist within a network, as well as peers to participate who may be behind firewalls or NAT systems and therefore do not have accessible network addresses. For a pipe to be created, the actual endpoints of peers need to be resolved. P2PS uses an EndpointResolver interface to represent a service that is capable of resolving certain endpoints. Pipes are generally unidirectional. The data is retrieved from a pipe by adding an entity as listener to the pipe. This listener is notified when data arrives.

2) P2PS peers use XML advertisements to represent the various services available to the network and corresponding queries to discover these services. Services are represented by ServiceAdvertisements. These are collections of PipeAdvertisements - adverts that represent pipes. The publish and discovery mechanisms follow a typical P2P pattern. Peers broadcast their advertisements or queries to other peers in their group. When a peer receives a query it checks its local cache to see if it has a match. If it has, it returns an advertisement for a pipe which can be used to connect to the service. Rendezvous peers are peers that act as gateways to groups of peers. These peers cache advertisements and also propagate queries to other rendezvous peers they know about. In this way queries can be disseminated among other groups via their rendezvous peer.

In order to bind a Web service infrastructure to these mechanisms we must do two things:

1) Enable peers to publish and discover service interface definitions, in effect endowing the system with Web service standards of service description.

2) Address the unidirectionality of the pipe mechanisms to allow for a request/response model between peers.

The first issue is dealt with by extending the advertisement structure used by peers. In the default P2PS implementation a ServiceAdvertisement is simply a collection of named PipeAdvertisements. A P2PS PipeAdvertisement is essentially a named endpoint - although the endpoint is logical and requires an EndpointResolver to turn it into a physical address. There is no mechanism for defining the data types or encoding expected by the pipe. In other words, there is no facility for service description beyond naming available pipes. Therefore we add a 'definition pipe' to our service adverts, from which the service definition (WSDL in our case) can be retrieved. We also specify the advertisements as schema allowing us to place them into a SOAP envelope.
This gives them the flexibility to contain arbitrary meta-data serialized as XML in the SOAP envelope’s header.

To handle the uni-directionality of P2PS pipes and allow peers providing services to return data from a request we make use of the WS-Addressing [30] specification. WS-Addressing defines an EndpointReference object and a means of serializing this reference to XML. An EndpointReference is an abstract definition of an endpoint. It has a mandatory Address field that contains a Uniform Resource Identifier (URI). It has a number of other fields that are optional, one of which is of interest to us - an extensibility element called ReferenceProperties that can contain arbitrary protocol or application defined properties to be set on the endpoint information we must specify a mapping between P2PS pipe advertisements and WS-Addressing EndpointReferences as these are the two data structures that represent endpoints. In order to achieve this we first define a protocol we call ‘p2ps’, which uses the URI structure shown below:

\[\text{p2ps://169.254.25.129-1092304889867/Echo#echoString}\]

1) The Address field contains the peer id followed by the name of the ServiceAdvertisement that the pipe is associated with, as a URI. If there is no service associated with the pipe, for example a pipe that is merely a return channel for an invocation, then a service name is omitted. In this case the Address field is just the scheme and the host component, i.e. the peer id.

2) The EndpointReference contains a set of ReferenceProperties that are the other fields of the advert, including the name of the pipe.

3) A SOAP invocation directed at a P2PS peer pipe must contain a SOAP header representing the pipe advertisement that advertises the pipe being addressed according to the rules of WS-Addressing. In other words, the To field becomes the URI contained in the Address field of the EndpointReference and the Action field becomes the Address URI appended by a fragment component that represents the pipe name. The ReferenceProperties of the EndpointReference are copied directly into the SOAP header (as specified by the WS-Addressing SOAP binding).

4) This SOAP header can contain a ReplyTo field which defines the endpoint (pipe advertisement) to send a response to.

5) This SOAP header can contain any of the other optional WS-Addressing fields.

Figures 5 and 6 show the processes involved in sending and receiving a SOAP message which contains a PipeAdvertisement serialized as an EndpointReference. Before a consumer peer sends a SOAP message down a pipe, it asks P2PS to create a new pipe - which allows incoming messages - and a corresponding PipeAdvertisement. The peer adds itself as a listener to this pipe. It then serialises the PipeAdvertisement to an EndpointReference which it includes in the header of its SOAP request. At the service provider end, the peer converts this reference to a PipeAdvertisement which it then requests P2PS to resolve to a pipe - the pipe created by the consumer peer. Once processing is complete the data can be returned to the consumer peer.

When a message arrives down the response pipe the is data extracted. The Client is then notified, which in turn notifies the listening application.

V. WSPeer Applications

WSPeer has been used in a number of application scenarios. As stated, the motivation behind WSPeer was to research the cross-fertilisation of Web services and P2P systems. One of the key deliverables of this research was to provide a stable generic Web service platform for the Triana problem solving environment [31]. From the Web services perspective, Triana provides a convenient graphical environment in which to discover, invoke and choreograph Web services. Users discover and search for Web services by quizzing repositories (e.g., UDDI) or searching through P2P networks for WSDL files.
When the matching Web services are located, they appear as standard tools within a Triana toolbox. Users can drag these icons onto a scratchpad and wire them together to create Web service workflows. WSPeer is being used by Triana to perform all of its Web service capabilities [32]. Triana, in turn, is being used by several projects worldwide.

The facility for dynamically defining and deploying Web services at run time within WSPeer’s light-weight hosting environment (as described in Section III) has recently been applied to a Grid computing scenario. In the scenario, a Triana unit was created that used WSPeer to launch a Web service, having first launched a Cactus [33] simulation on a distributed resource. Cactus generated output files (e.g. JPEGs) which showed state changes during the solving of a hyperbolic partial differential equation using finite differences. These were passed back to Triana via the WSPeer generated Web service in real-time as the simulation iterated through its time steps. The scenario was demonstrated at Supercomputing 2004 in the Center for Computation and Technology at Louisiana State University (CCT LSU) booth.

The P2PS implementation of WSPeer is currently being evaluated by the Catnets [34] project as a potential application platform for exploring how economy driven services interact in a decentralised topology.

VI. CONCLUSION

Current incarnations of Web services and P2P systems suffer from distinct and, in many cases complimentary drawbacks. While Web services are strong on standards, they show possible weaknesses on issues such as scalability and resilience to node failure. P2P systems on the other hand are strong on these issues, yet lack open, shared standards of service definition and message exchange. We believe that by combining the strengths of both, we can create more viable, flexible and ubiquitous service deployment scenarios. To this end we have been developing WSPeer. By producing such distinctly different initial implementations of WSPeer we have shown how Web services can be deployed into and invoked from vastly different environments and still pass through the same homogenising layer to a controlling application. This layer allows applications to exploit the diversification of service deployments while at the same time remain protected from this very diversity, allowing faster, more responsive development to change.

More information on WSPeer can be found at http://www.wspeer.org.

REFERENCES