Dynamic Web Service Deployment Using WSPeer

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

Dr Ian J. Taylor
School of Computer Science,
Cardiff University

Abstract— **This paper describes how WSPeer, a framework for deploying and invoking Web services, allows application code to host and invoke Web services. We show how the mechanisms used by WSPeer shield applications from implementation changes and enable applications to easily expose elements of themselves as Web services. The mechanisms described differ from usual Web service deployment scenarios in which a service is deployed into a container framework which manages the service environment. Instead WSPeer leaves control in the hands of the application. This allows Web services to be created and deployed that can respond dynamically to application requirements.**

I. Introduction

WSPeer is a framework for hosting and invoking Web services allowing application code to act as both service provider and consumer with potentially varying and evolving Web service architectures while maintaining a consistent interface. Unlike many other Web service infrastructures however [1] [2] [3], WSPeer is not built on the container model usually associated with Web service hosting frameworks. The container model establishes an environment into which software components are deployed as services and manages the lifecycle and requests for the services. This model is useful for managing potentially large numbers of applications and is therefore used by organisations and businesses who want to expose multiple applications via a single endpoint. As a consequence it is generally applied to client/server architectures. Because Web services are usually deployed in this context, they have inherited the trappings of this client/server approach, in particular the use of HTTP as a communication protocol and Universal Discovery, Description and Integration (UDDI) for service discovery. These protocols are suited to client/server architectures. An HTTP server requires a accessible IP address which a network entity cannot always supply if, for example, it is behind a Network Address Translation (NAT) system. Furthermore this address is usually presumed to be static and permanently available, making it impossible for nodes that are assigned addresses dynamically or wish to join and leave the network at their discretion, from behaving as service providers. The use of UDDI as a discovery protocol is also client/server centric in that UDDI registries are dedicated entities in the network. Furthermore, like web servers, UDDI registries are expected to be persistent and always available. Apart from precluding certain types of nodes from behaving as service providers and discovery registries, this centralised approach also scales badly because increasing the number of nodes in the network can cause performance bottlenecks [4] [5].

WSPeer has evolved out of an interest in combining the strengths of Web services - in particular the XML technologies of Web Service Description Language (WSDL) [6] and Simple Object Access Protocol (SOAP) [7] - with the strengths of Peer-to-Peer (P2P) systems - decentralised resource sharing and discovery mechanisms. Web service XML standards allow interoperable service definition and message exchange while decentralised resource sharing and discovery have proven themselves to be scalable and resilient to node failure while enabling transient nodes to interact. This concern is reflected in the architecture of WSPeer. Instead of requiring applications to embed themselves within a container, WSPeer acts as an interface to remote service consumers and providers, leaving control with the application. This architecture enables applications to easily expose themselves, or parts of themselves, as Web services. We call this facility *dynamic deployment* because the application remains in control of the service, that is, the service remains in the application environment. This in turn allows services to respond dynamically to the requirements of the application. Combining this strategy with decentralised mechanisms of discovery and resource distribution exemplified by such P2P systems as Gnutella [8], Chord [9], CAN [10] and Pastry [11], opens the way for Web services to be deployed as transient interfaces to P2P style peers. Furthermore, WSPeer acts as a buffer between the application and the underlying detail of the network. This enables an application to dynamically traverse networks at runtime.

This paper describes the mechanisms used by WSPeer to deploy and invoke Web services showing firstly how these mechanisms shield applications from implementation changes and secondly how they enable Web services to be used in more transient contexts. Section II gives a brief overview of WSPeer showing the process by which applications invoke a remote service and are notified of events via WSPeer's event structure. Section III discusses the two types of service deployment supported by WSPeer. Section IV then describes a simple example of a chat application implemented using Web services. Although this example is trivial it displays the essence of how WSPeer creates dynamic Web services.

II. WSPeer Overview

WSPeer is constructed as a tree of interfaces. Figure 1 shows this structure.

The `Server` interface represents the service provider aspect of WSPeer and the `Client` interface represents the

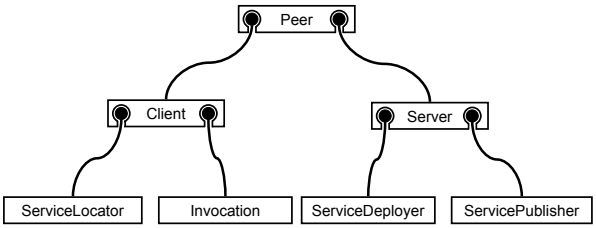

Fig. 1. WSPeer Interface Tree

service consumer aspect. On the provider 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 service. Publishing the service involves making the service endpoint and/or its interface description available to the network in some way. The consumer side of the tree is responsible for locating services and subsequently invoking them. The components in the tree are pluggable, allowing application code to replace or add new nodes at any level of the tree, either as part of the configuration of the application or at runtime. The latter approach enables an application to deploy, publish or discover services in multiple network scenarios concurrently. Nodes in the tree receive notification of events fired by their child nodes. All events are propagated upwards to the root of the tree - the `Peer` interface. Application code implementing the `PeerMessageListener` interface adds itself as an event listener to the `Peer`. Below is a code listing of the methods specified by the listener interface.

```
public interface PeerMessageListener {

  public void peerMessageReceived(
              DiscoveryMessageEvent evt);
  public void peerMessageReceived(
              PublishMessageEvent evt);
  public void peerMessageReceived(
              ClientMessageEvent evt);
  public void peerMessageReceived(
              ServerMessageEvent evt);
  public void peerMessageReceived(
              DeploymentMessageEvent evt);

}
```

When an application requests WSPeer to deploy a service, it receives a `DeploymentMessageEvent` on completion. When WSPeer publishes a service the application receives a `PublishMessageEvent`. When the service is invoked, the application receives two `ServerMessageEvent`s either side of the service being invoked. The first specifies it is an incoming event and allows the application to do any pre-processing before requesting the server to invoke the service. This might involve performing security checks or processing contextual information related to the service. This pre-processing may of course result in the service invocation being abandoned. The second event is specified as an outgoing event and allows the application to perform post-processing, for example updating state as a result of the service invocation, before the server sends the response (if there is one) back to the service consumer. This event structure gives the application control over service invocation. An application need not act on every event it receives. For example if it is not interested in processing server-side messages, it can ignore the event.

Service discovery is achieved through the `ServiceLocator` interface. This interface receives a `ServiceQuery` object from the application with which to locate services. A `ServiceQuery` is used by WSPeer to represent criteria for matching a service. If a match is found, then the application is notified with a `DiscoveryMessage`. This message contains a `ServiceDescription` object which has been generated from a service description document, WSDL for example. However a service itself cannot be invoked - rather it contains one or more invocable operations represented in WSPeer by the `OperationDescription` object. Therefore the application must inspect the service description to discover the available operations before it can invoke anything. This can done using an `OperationQuery` object. An `OperationQuery` contains logic to return an `OperationDescription` object given a `ServiceDescription`. Finally, armed with an `OperationDescription`, the application can retrieve an `Invocation` object. An `Invocation` uses the data in an `OperationDescription` - inputs, outputs, endpoint etc - to invoke the remote service operation via its `invoke(Object[])` method. If data is returned from the invocation, then the application is notified via a `ClientMessageEvent`.

As we can see from the above descriptions, an application is shielded from underlying implementation detail because it deals with WSPeer specific data types and events. This allows implementations to be changed without affecting the application layer. This is important, not only from an application development perspective but in order to allow applications to dynamically cross between different networks at runtime.

There are currently two implementations of WSPeer. The first uses standard Web service technologies - HTTP as a transport protocol and UDDI as a publish and discovery protocol. The second uses a P2P API called Peer-to-Peer Simplified (P2PS) [12]. P2PS uses *pipes* for communication. Pipes are abstract channels that can span multiple nodes and transport protocols. These channels have listeners attached to the receiving end which are notified if data arrives. Publishing and discovering services in P2PS currently uses two mechanisms - firstly a broadcast mechanism within groups of peers and secondly via *Rendezvous Peers*. Rendezvous peers cache service advertisements and propagate service queries to other peer groups.

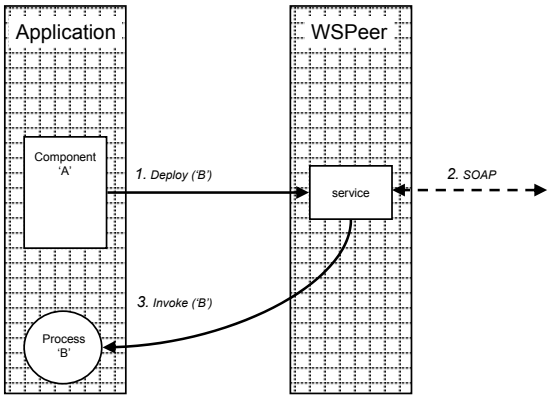

Fig. 2. Direct Deployment

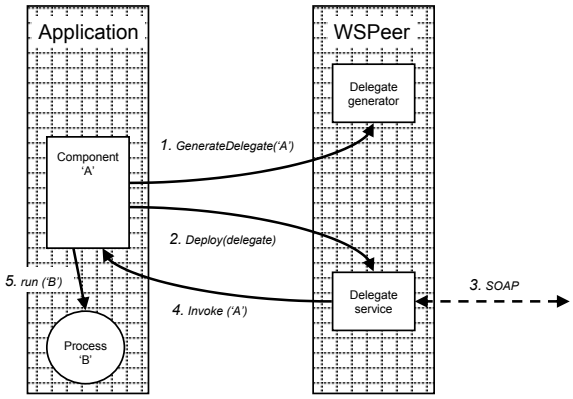

Fig. 3. Delegated Deployment

III. Service Deployment

WSPeer supports two types of service deployment. The first we call *direct* deployment. In this scenario the application passes WSPeer a Java class that should act as a front-end to a process. WSPeer deploys the class as a service (using Apache's Axis [13]) and then establishes an endpoint at which the service can be connected to. In the case of the standard WSPeer implementation, this is a light-weight HTTP server limited to retrieving and listing deployed services. In the P2PS implementation this endpoint is a pipe which has a listener component attached to it. Figure 2 shows the processes involved in this deployment scenario. The significant departure from the container model here is that the component being deployed does not leave the application environment. When the service is invoked, the call gets directed to the component within the application.

The second type of deployment we call *delegated* deployment. Here a delegate, or proxy, component is generated at runtime which passes calls back to an object in memory. The delegate is the component that gets deployed. The object that the delegate calls back to might be the application itself, or a component within it that is capable of initialising or steering sub-processes within the application. Figure 3 shows the processes involved in this scenario.

Here the application calls on the `DelegateGenerator` which is capable of creating a Java class that will be deployed as a service. In order to generate this class the `DelegateGenerator` takes a reference to an object that the service should delegate its invocation to. It can then be requested to create operations for the service which map to function calls on the referenced object. In Figure 3, a component passes itself as the object reference to the `DelegateGenerator`. When the service is invoked, the generated class passes the calls back to the component. This in turn initialises a process elsewhere in the application.

The `DelegateGenerator` takes a `DelegateEncoder` instance which performs the mapping. The most simple encoder simply maps input and output parameters back to the object. More complex encoders perform transformations to the received data before calling the object, allowing the service operation to differ from the function call. The generated class file simply performs the encoding and then performs a static lookup to find the object reference that is the target of the operation invocation.

The important difference from the direct deployment scenario is that here the service, although itself stateless, invokes an object in the application whose life-cycle is independent of the life-cycle of the service and which therefore may maintain state across service invocations.

The delegated deployment pattern is used by the problem solving and workflow environment Triana [14]. Triana employs an application-level protocol called the GAP [15] to act as an interface to various Grid and P2P architectures. WSPeer is used as the Web service binding to the GAP. The GAP uses the delegated deployment mechanism to create Web services that can call back to the GAP and hence to the Triana environment.

IV. The 'SoapBox' Service

This section describes a simple chat application implemented by a Java class called `SoapBoxClient`. This class exposes a user as a Web service called 'SoapBox' with a single operation called `receive`. This operation takes a string as input which is the remote peer's chat. The operation is invoked by remote users to send their messages. The `SoapBoxClient` class uses the *delegated* deployment model because the application must maintain its state across service invocations to allow for conversation. The `SoapBoxClient` class implements the `PeerMessageListener` interface, allowing it to receive messages from the WSPeer system. It has three public fields:

```
public static final String SERVICE = "SoapBox";
public static final String OP_NAME = "receive";
```

```
public static final String HELLO =
        "SoapBox speaker is online ";
```

The first is the name of the services we want to discover. The second is the operation name we wish to invoke on the remote services. The third is an introductory message which a client sends to a newly discovered service. When a client receives this message, it triggers a new search for the service.

The application also has a number of member variables.

```
private String user;
private Vector endpoints = new Vector();
private Peer peer;
private String localEndpoint;
// Cached operation description
private OperationDescription opdesc = null;
```

The first is the name we will use to identify ourselves to other SoapBox applications. The second is a list of the endpoints of remote SoapBox services. The third is a reference to a WSPeer `Peer` object. We also cache our local service endpoint. This allows us to check that we are not invoking ourselves. Finally we cache an `OperationDescription` object. We are able to cache this because we are only ever looking for one operation - the `receive` operation exposed by the SoapBox service. Hence when we build an `Invocation` object to invoke the services, we simply pass in the same `OperationDescription` and change the endpoint.

The class constructor initialises the chat application by assigning the user name and adding itself as a listener to the WSPeer event structure.

```
public SoapBoxClient(String user) {
    this.user = user;
    this.peer = PeerFactory.newPeer(
            PeerFactory.STANDARD_PEER);
    peer.addPeerMessageListener(this);
```

Next we create the service, using the `DelegateGenerator`, and then deploy and publish it.

```
DelegateGenerator generator =
    new DelegateGenerator(this);
generator.createService(SERVICE);
generator.createOperation(OP_NAME,
    new String[]{"java.lang.String"}, null);
ServiceSource source = generator.generate();
peer.getServer().getServiceDeployer().
    deployService(source);
peer.getServer().getServicePublisher().
    publishService(source);
```

The generator's `createService` method initialises a Java class with the given name (in this case 'SoapBox'). The `createOperation` method takes a name, a string array of parameter types and a string for the return type, or null if the method returns nothing. In this simple case the name and parameters map to an operation in the `SoapBoxClient` class of the same name and parameter types. This method - the `receive` method - is shown below.

```
public void receive(String s) {
    System.out.println(s);
    if (s.startsWith(HELLO)) {
        locate();
    }
}
```

It is the one method of the SoapBox service and is called by remote peers. As we see from the code listing above, it simply prints out the remote peer's message. If the message begins with the introductory message, then it attempts to locate the new service. A more efficient means of discovering newly joined clients would be for the introductory message to contain the endpoint of the client service. This could then be added directly to the list of endpoints cached by the application. Alternatively this could be done using WS-Addressing [16] headers in the invocations. With this mechanism each service invocation might contain the sender's reply address which could be extracted and added to the list of endpoints.

The `DelegateGenerator`'s `generate` method returns a `ServiceSource` object. This object is used by WSPeer to maintain meta-data about a deployed component. In the code above, once the `ServiceSource` is generated it is passed, first to the peer's deployer for deployment, and then to the publisher, to be published to the network.

A further step to initialising the client is to enable the client to send messages. For brevity, this is not listed here. It consists of a thread that reads from the command line. When the user hits return, the application's `send` method is invoked (see below).

The final step in initialising the client is to locate some remote services. This method is called once at initialisation time, and subsequently when a client is contacted by a service that has just joined the network and discovered existing chat services.

```
public void locate() {
    ServiceQuery query =
        new NameServiceQuery(SERVICE);
    peer.getClient().getServiceLocator().
        locateService(query);
}
```

Because the `SoapBoxClient` class implements the `PeerMessageListener` interface, it will be notified of events from WSPeer. The events it is interested in are deployment events (when it deploys its own chat service) and discovery events (when remote chat services are located as a result of calling the `locateService` method of the `ServiceLocator`, as in the method above. We look at these two methods in turn.

```
public void peerMessageReceived(
    DeploymentMessageEvent evt) {
    this.localEndpoint = evt.getMessage().
        getServiceSource().getEndpointURI();
}
```

When a deployment message is received, we assign our instance variable `localEndpoint` to the endpoint of our deployed service. This allows us to filter out our own service when communicating with remote services.

```
public void peerMessageReceived(
    DiscoveryMessageEvent evt) {
    ServiceDescription description =
```

```
            evt.getMessage().getQueryResult().
            getServiceDescription();
        OperationQuery opquery =
            new TypesOperationQuery(OP_NAME,
            new String[]{"java.lang.String"});
        opdesc = opquery.match(description);
        if (opdesc == null) {
            return;
        }
        String endpoint = opdesc.getPort().
            getLocationURI();
        if (endpoint.equals(localEndpoint)) {
            return;
        }
        for(int i=0; i<endpoints.size(); i++) {
            String endpoint =
                (String)endpoints.get(i);
            if (loc.equals(endpoint)) {
                return;
            }
        }
        endpoints.add(endpoint);
        send(HELLO + user, endpoint);
    }
```

When a service is discovered we need to do a number of things:

1) We extract a `ServiceDescription` object from the message. A service description is generated from a description document like WSDL.
2) Look for an operation called `receive` that takes a String as parameter in the `ServiceDescription`. We use an `OperationQuery` to perform this task. The `TypesOperationQuery` used above looks for operations with a specified name and certain input parameters. The `match` method returns an `OperationDescription`. We set our instance variable `opdesc` to be the returned `OperationDescription` from the `match` method. If the `OperationDescription` is null, then the service is not a real SoapBox and therefore we bail out of the method. Otherwise we extract the endpoint from the `OperationDescription`. If this endpoint is the same as our local endpoint then we have discovered ourselves. In this case we do not add the endpoint to our list of endpoints. If the `OperationDescription` represents a remote service, we add to our list if it has not been already.
3) Finally we send a 'HELLO' message appended by our user name to the newly discovered service.

The other methods of the `PeerMessageListener` interface remain unimplemented.

Finally we look at the `send` method of the `SoapBoxClient` class. This method invokes the `receive` operation of remote services. There are actually two `send` methods - the first takes the message to send as a parameter. This method just loops through the list of cached endpoints and calls the second `send` method which takes the message as well as the endpoint to send to as an argument. Below we list both methods. In both operations, if our locally cached `OperationDescription` is null, then we have not discovered anyone to talk to, apart from possibly ourselves.

```
    // send a message to all endpoints.
    public void send(String s) {
        s = user + ": " + s;
        if (opdesc == null) {
            return;
        }
        for(int i=0; i<endpoints.size(); i++) {
            String endpoint =
                (String)endpoints.get(i);
            send(s, endpoint);
        }
    }

    // send a message to a particular endpoint.
    private void send(String s, String endpoint) {
        if (opdesc == null) {
            return;
        }
        Invocation inv = peer.getClient().
                getInvocation(opdesc, endpoint);
        inv.invoke(new Object[]{s});
    }
```

We should note from the code listings above that the actual discovery and publish mechanisms lie below the application level and are implementation specific. The `SoapBoxClient` uses HTTP and UDDI because the `PeerFactory` is asked to create a standard peer:

```
peer = PeerFactory.newPeer(
        PeerFactory.STANDARD_PEER);
```

if we changed this single line to:

```
peer = PeerFactory.newPeer(
        PeerFactory.P2PS_PEER);
```

we would be using the P2PS implementation. It should also be noted that implementations need not remain self-contained. For example a UDDI enabled service locator could easily be employed by the P2PS peer.

Finally let us look at the generated code of the delegate service - the Java class that passes the call back to the `SoapBoxClient` class:

```
public void receive(java.lang.String in0)
        throws RemoteException {
    SoapBoxClient inst = null;
    try {
        inst = (SoapBoxClient)InstanceTable.
        getInstanceTable().
            get("1103048238871-36807");
    } catch(ClassCastException e) {
        throw new RemoteException(
            "Object is of wrong type.");
    }
    if(inst == null) {
        throw new RemoteException(
            "Object cannot be found.");
    }
    inst.receive(in0);
}
```

Essentially this class simply performs a lookup operation. The key is generated at the time the class is generated and the object instance passed to the `DelegateGenerator`'s constructor is placed in a static table with the generated key. The class then invokes the appropriate method on the object instance.

It should be noted that although we show source code for the delegate service, this class is, by default, generated directly as a byte array which is read by its own class loader. This bypasses the need for source code generation and compilation. WSPeer generates all its stubs directly as byte arrays as well, so all the remote services being discovered above never end up written to file. This is particularly useful in P2P scenarios where service persistence is not expected. Of course users can choose to generate and/or save their stubs and delegate services as Java class files if they so wish.

V. Conclusion

In this paper we have described how WSPeer allows application code to expose elements of itself as Web services. The mechanisms described represent a departure from standard Web service deployment scenarios which make use of containers to manage collections of services. WSPeer allows an application to maintain control over service life-cycle, state and invocation by adding service capabilities to the application rather than requiring the application to be deployed into a container environment. This makes WSPeer an ideal platform for extending the possibilities of Web service deployment and invocation. In particular it enables Web services to be used in P2P contexts.

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

References

[1] "Apache Jakarta Tomcat." [Online]. Available: http://jakarta.apache.org/tomcat/

[2] "IBM Websphere." [Online]. Available: http://www-306.ibm.com/software/websphere/

[3] "Sun Java System Application Server." [Online]. Available: http://www.sun.com/software/products/appsrvr/home_appsrvr.xml

[4] M. P. Papazoglou, B. J. Kramer, and J. Yang, "Leveraging Web-services and Peer-to-Peer Networks," in *Proceedings of the 15th Int. Conf. on Advanced Information Systems Engineering (CAiSE 2003)*, 2003, pp. 485–501.

[5] F. Forster and H. D. Meer, "Discovery of Web Services with a P2P Network," in *Computational Science - ICCS 2004: 4th International Conference, Krakw, Poland, June 6-9, 2004, Proceedings, Part III*, M. Bubak *et al.*, Eds., vol. 3038 / 2004. Springer-Verlag Heidelberg, 2004, pp. 90 – 97.

[6] W3C, "Web Services Description Language (WSDL) 1.1," W3C, Tech. Rep., 2001. [Online]. Available: http://www.w3.org/TR/wsdl

[7] W3C, "Simple Object Access Protocol (SOAP) 1.2," W3C, Tech. Rep., 2003. [Online]. Available: http://www.w3.org/TR/soap/

[8] "Gnutella." [Online]. Available: http://www.gnutella.com

[9] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," in *Proceedings of the 2001 ACM SIGCOMM Conference*, San Diego, California, USA, 2001, pp. 149–160.

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A Scalable Content Addressable Network," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, San Diego, California, USA, 2001, p. 161172.

[11] A. Rowstron and P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems," in *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, 2001.

[12] I. Wang, "P2PS (Peer-to-Peer Simplified)," in *Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, 2005.

[13] "Apache Axis." [Online]. Available: http://ws.apache.org/axis/

[14] "The Triana Project." [Online]. Available: http://www.trianacode.org

[15] I. Taylor, M. Shields, I. Wang, and O. Rana, "Triana Applications within Grid Computing and Peer to Peer Environments," *Journal of Grid Computing*, vol. 1, no. 2, pp. 199–217, 2003. [Online]. Available: http://journals.kluweronline.com/article.asp?PIPS=5269002

[16] A. Bosworth *et al.*, "Web Services Addressing (WS-Addressing)," Mar 2003. [Online]. Available: http://msdn.microsoft.com/ws/2003/03/ws-addressing/