State Aware WSDL

Michael Brock and Andrzej Goscinski

Grid and Service Oriented Architecture Research Group
School of Engineering and Information Technology, Deakin University
Pigdons Road, Waurn Ponds, Australia, 3217

{mrab, ang}@deakin.edu.au
http://www.deakin.edu.au/scitech/sit/dsapp/

Abstract

With the recent innovations in stateful web services, they are now being used to support the construction of distributed systems using software as a service. While the state of web services is preserved, the state is still hidden from clients thus searches for both functionality and state remains a two step process. Proposed in this report is the Resources Via Web Instances (RVWI) framework. RVWI grants to web services the ability to include their state and characteristics in their WSDL. This was done by allowing snapshots (instances) of a web service to be listed in the WSDL of the web service. Instances were utilised as they contain state and characteristic information directly from the web service. Thanks to the inclusion of state and characteristics, queries for web services can now be carried out on the availability of a web service and the 'dimensions' of resources.

Keywords: Web Services, Stateful Web Services, WSDL, Web Service Discovery, Web Service Selection

1. Introduction

A recent trend in distributed computing is the use of web services in software as a service. Instead of building software systems as huge 'monoliths' software systems are broken into several 'services'. Each service exposes a resource (be it a business process, data store, etc) to the World Wide Web and consumers (clients) make use of the resource via the service. It is even possible to build larger services from a set of smaller services.

Clients learn how to use the web service by requesting the WSDL. The WSDL is a XML Document written in the Web Service Description Language (Christensen et al., 2001). Through the WSDL, clients learn everything functional about the web service. This includes (among other things) the methods, data types and messages needed to communicate with the web service.

The WSDL is even used by discovery services, such as the Universal Discovery, Description and Integration (UDDI) (Bryan et al., 2002) Service. When a client requires a web service of certain functionalities, the client submits a list of functionalities to the UDDI services. The UDDI service compares the functional requirements against the WSDL of services it's aware of and then returns a matching service.

Web Services are able to keep state about themselves for long periods of time thanks to the Web Service Resource Framework (WSRF) (Czajkowski et al., 2004a). Previously, web services discarded state at the end of every client request. Thus, at the time, it wasn't easy to build distributed systems where requests depended on the outcomes of previous requests. WSRF furthered the use of web services by allowing web services to retain their state between client requests. WSRF preserved state by keeping all state information in a decoupled entity called an Instance. At the end of each request, the Instance (which contained the state of the service) was written to a database (or any other form of persistent storage).

When a request from the same client was received, the related Instance was loaded from the database into the service and the request performed. The Instance was always kept on the database until the client notified the Service that it was finished which resulted in the Instance being removed from the database. If the client doesn't indicate it's finished with the Instance, the Instance is eventually removed from the database by routine maintenance.

To tell which request belongs to what Instance, a unique identifier is created with each Instance. This identifier is called the Endpoint Reference (EPR) and is a simple string of letters and numbers. After an Instance is created, the client is given the EPR. When the client later makes additional requests, the client simply supplies the EPR, so the Service retrieves the correct Instance.

Thanks to the EPR, there is no limit to the number of Instances a web service can have. It is possible to have an Instance for each client that approaches the web service. It is also possible to keep a single Instance to all clients by simply passing the one EPR to all clients.

In spite the innovations in standards and technology for web services, little has been considered in adding state information into the WSDL. This means that a client cannot tell if a web service is busy (or even active) until attempts are made to access the web service.

Most standards and technology are focused on the inclusion of the functional aspects of web services and have left the discovery of state information to 'manual labour'; where clients first learn of a web service and then invoke methods to discover the internal state. This 'two-step' approach is ineffective as clients waste time on

trying to access services which don't meet expectations or no longer exist.

Proposed in this document is the innovative Resources Via Web Instances (RVWI) Framework for including state and characteristics of web services in their WSDL. As WSRF has already addressed making instances persistent, RVWI made further use of the instances by including them in the WSDL. Instances are snapshots of web services and have direct information about a web service's state and characteristics within them.

By allowing clients to see state and characteristic information in the WSDL, clients get a better view of the quality of the web service and see at an early stage the activity of the Service. The significance of RVWI is both Service state, characteristics and its Interface are now presented as a single unit. This gives clients a richer view of the web services they are examining. Instead of only seeing what a web service 'can' do, clients can now see 'what' the web service is doing.

Another major innovation of RVWI is the simplification of web service brokers. Instead of having large, complicated brokers which keep state information on services (via external services), brokers can now examine the WSDL of services and then decide which web service is the best match for the client.

2. Related Work

To learn of a web service, clients usually have to contact another service designed to find web services. Web services have the UDDI Service which keeps information on web services through Binding Templates (Bryan et al., 2002). The problem with UDDI is it doesn't keep information about the state of a given service. This means that if a service fails or becomes non-responsive, this change is not reflected in the UDDI Service. Nor is UDDI ever aware that any of its registered services have failed. UDDI is simply a registry, like a phonebook, thus UDDI doesn't concern itself with the activity of a service. (Ran, 2003, Sinclair et al., 2005) are two works which addressed the need for including non-functional aspects in UDDI registered web services.

In (Ran, 2003), a UDDI service was expanded to contain Quality of Service (QoS) characteristics of a service. (Ran, 2003) also proposed a new Service called the QoS Certifier, which certifies QoS claims from Service Providers about their Service. (Ran, 2003) enabled UDDI to find a service that both meet the functional needs of the client and one that did so effectively.

Instead of contacting the UDDI Service directly, the Service Provider first contacts the QoS Certifier Service and submits a QoS claim. The Certifier verifies the claim and downgrades if needed. The Certifier returns a Certificate ID which represents the Certified QoS Claim. The Service Provider registers itself to the UDDI Service using the Certificate ID and Service Information. The UDDI Service double checks the existence of the ID with the Certifier before accepting the registration.

While (Ran, 2003) has included support for queries based on QoS, the process itself still took two steps. The UDDI Services does return a set of services which fit the needs of the client but the client still needed to contact the Certifier about the QoS claims before contacting the Service. Client requests to the UDDI in (Ran, 2003) include both functional and QoS constraints. The UDDI Service returns a set of Services along with their Certificate ID. The Client verifies the Certificate IDs with the Certifier then makes use of one of the Services.

A similar work with UDDI is (Sinclair et al., 2005) where UDDI was expanded to support Grid Services. Grid Services are special evolutions of web services: their main differencing being suited to Grid environments and are capable of keeping state between requests. The preserved state in Grid Services was termed as Service Data and all information about the related Grid Service's current activity was stored in the Service Data.

(Sinclair et al., 2005) saw that it was possible to use UDDI to run searches on Grid Services by expanding UDDI to hold Service Data. When a Provider registered itself along with any Services, the expanded UDDI Service registered itself to the listed Services. In registering itself to the Services, changes in State were notified to the UDDI Service.

(Sinclair et al., 2005) was also the first example encountered during the research stages where queries for services and state are a one step process. In the case of (Sinclair et al., 2005) queries for both state and functionality were done in the one UDDI request. This meant when a client requested a service from UDDI, it received a service that fitted the functionality needs and was active at the time of the request and didn't need to be verified.

(Sinclair et al., 2005) and (Ran, 2003) both show it was possible to record non-functional aspects of Services via expansion of UDDI. This resulted in allowing clients to run queries for desired Services on more than just functionality. The only criticism of the two is both use UDDI thus access to QoS and State is not easy. Clients who contact the service cannot learn of the state directly. They can only learn service state either (i) using the methods the service provides (typical approach), or (ii) through the enhanced UDDI service.

In (D'Ambrogio, 2006) an attempt was made to include QoS information in the WSDL of a web service. Once exposed in the WSDL, it would have been easier to negotiate agreements between Clients and the service. This is different to (Ran, 2003, Sinclair et al., 2005) where the (respectively) QoS and state of a Service was kept in an external Service.

The main point about adding such information in the WSDL is the WSDL is the most freely available part of a web service. UDDI itself keeps a copy of the WSDL of a Service when a Provider registers itself. The reason for this is the WSDL is used for carrying out function requirement comparisons. Adding QoS specifications to the WSDL means that existing discovery services (such as UDDI) could also support QoS queries without (much) modification.

The only other two options for keeping the state information would have been either in a dedicated service (akin to MDS) or as a separate document that could be requested. The implication of using a separate service is failure in the service prevents the state from being accessible until the service is restored. The use of a separate document is plausible but additional effort is required to locate and obtain the document. Also such a document needs to be easily accessible and updated to keep consistent with the state of the service. Thus we keep the solution as simple as possible by keeping the information in the WSDL.

While there are solutions on listing non-functional aspects of Services, the focus is mainly on QoS factors. The interest is mostly on showing how quickly a Service can generate a response and so forth. Aside from (Sinclair et al., 2005) there is little focus on the publication of the state of a Service. Also the majority of the focus is on the use of external services such as UDDI.

3. RVWI Framework and the State Aware WSDL

The Resources Via Web Instances Framework (RVWI) grants unto web services the ability to show their state and characteristics in their WSDL. State has been included to show the current activity of the web service and characteristics have been included to show the nature of the resource(s) making up the web service. In RVWI, there are six main entities: the Resource, the Service, the Instance, the Application Server, the Provider and the Consumer. In RVWI, the owner of the Resources, Services, Application Servers and Instances is called the Provider and the end client, be a human or program, is called a Consumer. The Provider provides the Services and the mechanics needed to hold Instances of the given Services. The Consumer requests the use of Services from the Provider.

3.1. RVWI Entities

A Resource in RVWI is any resource that the Provider wishes to expose to Consumers via the Web. The Resource can be as simple as a text file or more complicated such as a computer.

The Service is a web service that exposes the Resource on the World Wide Web. The Service lists methods that can be used to manipulate and query the Resource. There is even the possibility of creating Resources via the Service but that matter is beyond the scope of this document. While the Service lists the operations, the operations are performed on Instances and rarely on the Service itself.

The Instance is an ‘active’ expression of the Service. When a Consumer makes a request to the Service, the request is sent to the Service which then performs operations on a retrieved Instance (Czajkowski et al., 2004b). Thanks to the EPR supplied by the Consumer, the Service can find and load the correct Instance.

The Instance is like a snapshot of a web service. It holds the current activity of the Service (State) with respect to the Consumer request. At the end of each request, the Instance is saved to database. When the same Consumer makes another request, the Service loads the Instance from the database and operations performed on it. An Instance also contains information on where Resources are located and what permissions are required to access the Resources. By keeping such information in the Instance, the Service can make use of the Resource and not have to learn multiple times how to use the Resource.

Finally, the Application Server is a ‘container’ that holds both Services and Instances. The Application Server provides an environment to help house Services, Instances and other resources. Application Servers also provide management operations, mainly addressing (where requests are directed to the correct Service). The Application Server is also capable of accessing other services, such as databases, when needed.

A Provider may have more than one Application Server thus may have an Application Server at multiple locations to improve performance (which is beyond the scope of this document thus not elaborated). The Application Server is also the point of contact with Consumers. When a Consumer makes a request, the request is processed by a Service in the Application Server. The Application Server then determines which Service the request is intended for and forwards the request on.

As stated before, Consumer requests are processed on an Instance, retrieved by the Service. If no Instance exists, a new one is created for the Consumer. The management of Instances (creation, destruction, etc) is handled by the Service, which approaches stateful resources and request their use (see (Czajkowski et al., 2004b)).

While Instances in (Czajkowski et al., 2004a) are created, stored and (eventually) destroyed they are not included when the WSDL of Services are requested. This is because WSDL is used to describe the functional aspects of a Service, mainly its operations. If a Consumer wants to know of the internal state of a Service, the Consumer needs to contact the Service directly and use operations listed in the WSDL. If not that, the Consumer needs to contact a service like UDDI which will learn of the state on behalf of the Consumer.

RVWI simplifies web service selection by allowing State and Characteristics of an Instance to be included in the WSDL. The reason for making Instances viewable in the WSDL is (i) it represents the Resource and (ii) the most vital factors in judging the quality of a service exist in the Instances.

3.2. Modifying the WSDL

Usually, when a Consumer requests the WSDL of a Service (Step 1 in Diagram 3.1), the request is accepted by the Application Server and then passed to the intended Service (2). Once the Service generates the WSDL (3), the Application Server transmits the WSDL to the Consumer (4).

RVWI adds one more step. Once the WSDL is generated (3), the Application Server checks with a database to see if there are any Instances of the Service (3a). If the Application Server finds any Instances, they are included into the WSDL (4). If the Application Server checks the

Server (3b) and cannot find any Instances of the Service, the WSDL is returned without any modification.

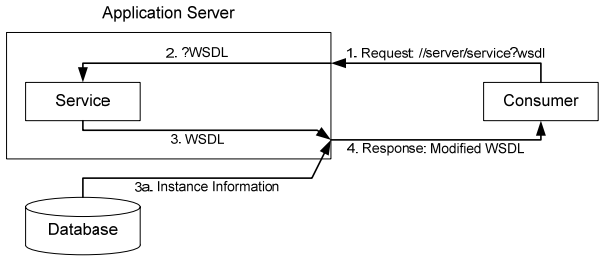

Diagram 3.1: Modifying the WSDL

The WSDL itself is only generated when requested of clients. This means that even if the service undergoes frequent change, the WSDL will only show the updated state when the WSDL is requested by the client. This is because the WSDL itself is unaware of the state until Step 3a in Diagram 3.1 is executed.

The information about Instances is retrieved from a Database. An Instance only exists in the Application Server when a Consumer request is being processed. Thus in Diagram 3.1 the Application Server gets the WSDL then 'annotates' it with information it finds in the database.

For Consumers, it is quick and easy to see the functionality and state of a web service. For brokers the inclusions of state and characteristics in the WSDL allows them to make more refined searches for web services. Thanks to the inclusion of state, brokers can now chose Services that are not busy and can promptly process client requests thus improving the response time of Consumer Requests.

The inclusion of characteristics further improves the selection of Services by allowing Consumers to specify the nature of the Resources they need. For example, a Consumer may want a file web service that can support files of a huge size. Currently, Consumers can request web services that support files, but cannot state a needed file size.

3.3. WSDL vs. XML

The WSDL of web services is an XML document. XML describes the structure of data and the rules behind the structure of the data. In XML data is converted to text, and 'wrapped up' in an XML element. The element contains information on what type the data is. For more complex data, like lists or tables, a XML element may contain a collection of child elements, each holding either data or more child elements.

XML has a useful feature called namespaces. Thanks to namespaces, XML elements can be grouped together. For example, there can be a namespace for XML elements that make up an array and one for elements that make up a list. Thanks to namespaces, elements with the same name cannot be confused. An XML document is usually made of one or more namespaces. All XML documents have a default namespace which element not allocated to a namespace are kept.

As the WSDL of web services contains a set of elements, RVWI works by adding an element of its own called the Instances element. The WSDL specifications (Christensen et al., 2001) lists the following elements of a WSDL document:

- **types** – describes the data types needed to invoke methods on the web service;
- **message** – defines a message that carried one or more types to be used on a method;
- **portType** – states a single operation of the web service. It also state one message that represents the request for the given operation and one message for states the response of the operation;
- **binding** – states how a request/response on a operation (portType) is transmitted over the network;
- **port** – the network address that the binding is to be invoked on. Just like how web pages have web addresses, operations in web services have them two;
- **service** – a collection of ports which make up the functional interface of the web service.

Each of the above XML elements describes details of the web service. Figure 3.2 shows a simple WSDL with a single method (some sections have been collapsed and other parts have been removed for simplicity).

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <schema targetNamespace="http://tempuri.org/">
      <element name="Increment">...</s:element>
      <element name="IncrementResponse">
        <complexType>
          <sequence>
            <element name="IncrementResult" type="s:int" />
          </sequence>
        </complexType>
      </element>
    </schema>
  </wsdl:types>
  <wsdl:message name="IncrementSoapIn">
    <wsdl:part name="parameters" element="tns:Increment" />
  </wsdl:message>
  <wsdl:message name="IncrementSoapOut">
    <wsdl:part name="parameters" element="tns:IncrementResponse" />
  </wsdl:message>
  <wsdl:portType name="CounterServiceSoap">
    <wsdl:operation name="Increment">
      <wsdl:input message="tns:IncrementSoapIn" />
      <wsdl:output message="tns:IncrementSoapOut" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="CounterServiceSoap"
        type="tns:CounterServiceSoap">
    <soap:binding transport=
          "http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="Increment">...</wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="CounterService">
    <wsdl:port name="CounterServiceSoap" binding=
          "tns:CounterServiceSoap">
      <soap:address location=
            "http://localhost/CounterService.asmx" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Figure 3.2: Example WSDL

WSDL has a section for each type stated in the WSDL specification. Each section describes a small unit of the web service and can call upon the section(s) before it to enrich its explanation. The PortType section, for example, states an operation thus one part of the web service has been described. Also, it refers to the messages sections on how to invoke the operation thus enriching its explanation.

For most of the elements, there is a prefix 'wsdl' that denotes a namespace. An element is related to a

namespace by looking at the prefix. If there is no prefix, an element is assumed to belong to the default namespace: a designated namespace where any element goes that doesn't have a namespace. With WSDL being modular (via sections) it was possible to add additional information about a web service by keeping it in its own section.

3.4. Modified WSDL Structure

The advantage of WSDL is it allows for additional information to be included provided that the additional information is kept it its own section. This make the additional information easily accessible while also preventing it from conflicting with other sections of the WSDL.

Figure 3.3 shows an example of a modified WSDL. All additional information provided by RVWI is kept in new section called *Instances*. This element keeps a list of InstanceInfo elements as its children.

```
<wsdl>
  <Instances>
    <InstanceInfo EPR="Instance EPR String">
      <State>
        <Description Name="ElementName">
          ElementValue</Description>
        ...Other Possible State Elements...
      <State>
      <Characteristics>
        <Description Name="CharacteristicName">
          CharacteristicValue</Description>
        ...Other Possible Characteristics...
      <Characteristics>
    </InstanceInfo>
    ...Other Possible Instances...
  </Instances>
  <types>...</types>
  <message name="MethodSoapIn">...
  <message name="MethodSoapOut">...
  <portType name="CounterServiceSoap">...
  <binding name="CounterServiceSoap">...
  <service name="CounterService">...
</definitions>
```

Figure 3.3: Modified WSDL

The InstanceInfo element holds information about a single Instance. This element only has one attribute, the EPR attribute. As the EPR is a unique identifier for an Instance thus has been included to uniquely identify each InstanceInfo element. Also, by including the EPR as an attribute, Consumers can quickly make use of the Instance by getting the value of the attribute. The InstanceInfo element only has two children, the State element and the Characteristics element.

In RVWI, state is represented as a collection of elements, each element having a name and value. This gives RVWI a high level of flexibility of showing state. For example, an Instance showing a text file will have two elements that make up the state. One element is to indicate if the file has been locked and another to indicate where the file is currently being read from or written to.

The State Element has one to many Description Elements to represent the elements that make up the state of the Instance. Description Elements basically are name value pairs with the name kept in the attribute and the value kept in the Element itself.

The Characteristics Element is almost identical to the State Element except for the role of the Description Elements. While in the State Element where each Description represented a part of the whole state, Description Elements in the Characteristics Element represent an individual characteristic. For example, the size of a file is characteristic that is kept in its own element.

The use of Description Elements gives a high level of granularity to describe the state and characteristics of an Instance. The reason for the high level of granularity is to ensure that an Instance can be described as best as possible. If the granularity is too low, client requirements cannot be best satisfied as there isn't enough information to refine the search.

3.5. Application Servers and Pipelines

Application Servers are like containers for web services. They provide the execution environment, such as CPU, memory, etc, and perform the delivery of requests from Consumers to the correct Service(s). Foremost, Application Servers are message processors and if a request cannot be delivered to a Service, the Application Server sends an error message back saying why. Application Servers also provide management functions for Services but they are beyond the scope of this document. Examples of Application Servers are Apache Tomcat (Apache Software Foundation, 2007) and Microsoft's ASP.Net (Corporation, 2007).

Application Servers are also capable of generating the WSDL to their contained Services. The WSDL of the Service is kept within the Application Server thus it has to be captured and modified before the Application Server sends it to the Consumer. Thankfully, Application Servers are modular so they can be customised to suit the needs of the Service Provider. Thus it is possible to add RVWI to an Application Server instead of having to build a whole new Application Server from the ground up.

The modularity of an Application Server is presented as a pipeline, a chain of modules that each performs a micro amount of processing on incoming requests. Pipelines are an extension to the message processing capabilities of the Application Server. After the Application Server determines the Service a request is intended for, the request goes through the pipeline, Diagram 3.2.

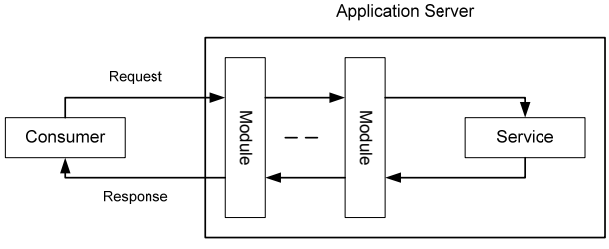

Diagram 3.2: Application Server Pipeline

Rather than complicate the Service to hold the full

validation process, the validation itself could be broken up into a series of modules and the modules then added to the pipeline of the Application Server. As there is a pipeline for each Service, modification to one pipeline doesn't always affect other services. The advantage of the pipeline is support for additional protocols and/or services is possible by wrapping the functionality into a module and inserting the module into the pipeline. This was done with RVWI where the functionality for modifying the service WSDL was wrapped up and inserted.

While attractive, modifying the pipeline can be difficult. As the pipeline stands between the Service and the Client Request, the pipeline must be as efficient as possible. Thus, the overall processing from all the modules in the pipeline must be minimal.

For example, if a module takes too long passing the request/response to the next module in the pipeline, the Consumer may give up waiting on the response. While the majority of works shown in Section 2 made use of an external process, RVWI focuses on the pipeline as it's a quick and simple way to modify WSDL responses.

Since RVWI seeks to list all existing instances of a service from the database, processing is kept in the pipeline. Another advantage of putting a module in the pipeline is the module automatically gains the set of permissions (security credentials) of the Application Server. This means if the Application Server has permissions to access a database, the module also gets the same permissions to the database. Had a full service been implemented, permissions would needed to been granted to the service so it could access the database.

3.6. RVWI Architecture

In order to modify the WSDL of a Service, a RVWI Module was proposed, created and added into the pipeline of the Application Server. All Instances in RVWI are able to write themselves into the database but it is the RVWI Module that retrieves them and places them in the WSDL. Diagram 3.3 shows the RVWI Module inside the Application Server's pipeline and its use of the Database.

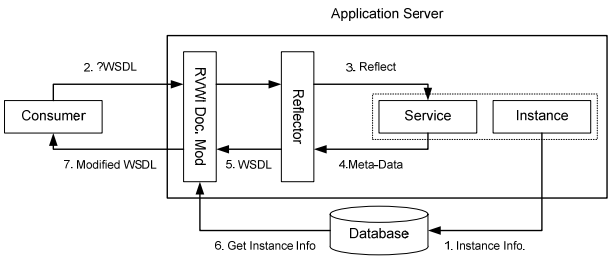

Diagram 3.3: RVWI Module and Database

The Instance, decoupled from the Service, keeps up to date information about itself in a Database (Step 1 in Diagram 3.3). Like how state is preserved in WSRF web services, fresh information is recorded to the database at the end of each request. An Instance is removed from the database when the Consumer signals to the Service it is finished with the Instance or the Instance doesn't get used for a prolonged period of time.

When a request for the Service WSDL is received (2), it passes through the pipeline. For this example, only the RVWI and Reflector modules are shown. After receiving the request, the Reflector module reflects the Service for meta-data (3). This meta-data includes the functional aspects of the Service and the required data types needed to make use of the Service. This meta-data is later used to make the resulting WSDL by taking the meta-data and transforming it using the WSDL Schema.

With this Meta-data (4), the Reflector generates the WSDL then tries to pass it back to the Consumer (5). Instead, the WSDL was intercepted by the RVWI Documentation Module (RVWI Doc. Mod.) which then queries for Instance Information about the given Service from the Database (6). For each Instance recorded in the Database, a XML Element was created and added to the WSDL. If there was no Instance Information in the Database, the RVWI Documentation Module didn't modify the WSDL. Either way, the WSDL document was returned when the RVWI Documentation Module was finished (7).

The end result is a WSDL that is aware of Instances of its Service. With the inclusion of state, brokers can now keep searches to active web services and the inclusion of characteristics, Consumers can now give characteristics on what a web service should have and be capable of supplying.

4. Implementation

RVWI was implemented as a class library, called the RVWI Service Library, using classes from WSRF.Net (Wasson, 2006) in C#. WSRF.Net is an implementation of WSRF (Humphery and Wasson, 2005) and was used to provide the recording of Instances to the database. WSRF.Net is a code library that provides all the support for stateful web services for programmers. Programmers are able to code as through coding a normal web service and WSRF.Net provides the rest in the background. C# was used as it was an easy to use language and support was more abundant than that of Java and C++. C# is also from the Microsoft .Net Framework (Microsoft Corporation, 2007) which has excellent support for the construction of web based systems.

The RVWI Service Library has a base class, RvwiServiceBase, for building stateful web services and the library itself has been divided into three 'packages': Description, Attributes and Modules. The Description package of the RVWI library contains all the classes needed to hold state and characteristic information when modifying the WSDL. The Attributes package contains a set of attribute classes to help annotate the web service. In C#, how a program is constructed can be influenced easily by using attributes.

Any Modules to be added to the pipelines of Application Servers is kept in the Modules Package. At the time of writing, RVWI only has the RVWI Documentation Module which finds Instances from a database and includes them in the WSDL. In future, should RVWI

grow to have more modules, they will be placed in this Package.

4.1. RVWI Service Library Classes

In WSRF.Net there are three main classes used to build stateful web services:

- SeviceSkeleton – provides the core functions needed to make a stateful web service.
- GCGResourceFactoryPortType – provides the functionality for creating Instances of the Service.
- ImmediateDestructionProxyPortType – provides the functionality for destroying Instances when the Consumer is finished with the Service.

RVWI acts as an extra layer between the Stateful Service and WSRF.Net. This is because WSRF.Net records Instances in such a way that it cannot be told which elements of the Instance makes up its state and which make up its characteristics.

The most vital class in RVWI is the RvwiServiceBase class. This class provides the mechanics for holding the state and the characteristics of the Service. Diagram 4.1 shows the relationship between RvwiServiceBase with the WSRF.Net classes.

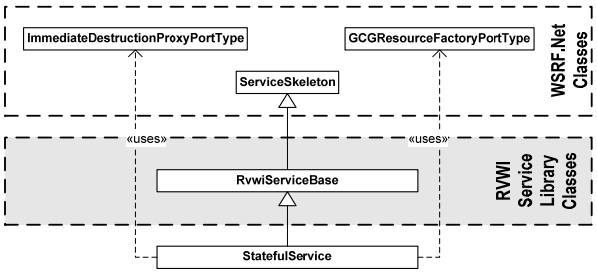

Diagram 4.1: RVWI building on WSRF.Net

Fortunately, the structure in which WSRF.Net keeps data is customisable. Thus, the RvwiServiceBase performs a conversion and WSRF.Net writes the converted structure to the database. The reason for converting the information to another structure was to minimise processing when modifying the WSDL.

As stated before, all processing in the pipeline needs to be kept minimal otherwise it could affect the performance of the Application Server. Thanks to the RvwiServiceBase class, the WSDL was modified by simply retrieving the Instance information from the database and pasting it directly to the WSDL.

4.2. Description Package

The Description Package contains all the classes responsible for describing an Instance of a web service. There are four classes in this package: InstanceDocumentation, InstanceInfo, ElementList and DescriptionElement. They are responsible for holding the state and characteristics of a Service Instance and ensure that they are serialized (converted to XML) to and from the database. They also implement the design of the RVWI XML elements from Figure 3.3

The InstanceDocumentation class represents the new 'Instances' section to be added to the WSDL. This class keeps a list of InstanceInfo objects, each on representing a single Instance of the Service.

The InstanceInfo class has two ElementList objects, one for state elements and one for characteristics. Other information about the Instance (such as the EPR) is also kept in this class.

The ElementList holds a collection of DescriptionElement objects. Each DescriptionElement holds one element of state or one characteristic. The DescriptionElement keeps the value and type of the state element or characteristic.

4.3. Attribute Package

The Attribute Package provides a set of Attributes which allow Service programmers (creators of web services) to annotate their Services. This annotation relates to indicating which elements of a Service Instance makes up the State and which elements are Characteristics.

Attributes are an advantage in .Net as they also allow programmers to influence how their programs are compiled and executed. Instead of having to write long complicated code, programmers annotate with attributes and .Net performs specific processing on the attributed elements.

RVWI has two attribute classes:

- InstanceState – an attribute that annotates the given property as an element of the Instance's state
- InstanceCharacteristic – an attribute that annotates the given property as a characteristic that characterises the Instance

To prevent the service programmer from manually indicating what elements make up the state and what elements make up the characteristics, the attributes are supplied so RVWI discover the elements automatically. By including these attributes, RVWI services are able to look at themselves and learn what elements of themselves make up the state and what make up their characteristics.

4.4. Modules Package

Modules in RVWI to be added to the pipleline of Application Servers are kept in this package. Currently there is only one module but the division is kept in the event that more modules are added to RVWI. In this section, there is only the RVWI Documenting Module.

Operation of the RVWI Documenting Module was simple: (i) capture the WSDL of the service as it was being sent to the client, (ii) include any instance information found from the database, and (ii) forward the modified WSDL to the client.

To get the RVWI Documentation Module into the pipeline, all the Service programmer has to do is modify the web.config file of an .Net web service to include the module, and .Net handles the inclusion of the module to the pipeline.

5. Feasibility of the Concept – The RVWI Counter

To test the use of RVWI, a simple Counter Service was built using RVWI (and WSRF.Net (Wasson, 2006) as RVWI sits on top of it). The idea of a Counter Service was used because it was simple and had both state and characteristics. The value of the counter was its state and the amount the counter was incremented and decremented by was the characteristic.

5.1. Design

Diagram 5.1 shows the class view of the Counter Service. The ServiceSkeleton class provides all the base methods needed for WSRF.Net web services. GCGResource-FactoryPortType and ImmediateDestructionProxyPort-Type are class used to provide methods for Instance management. The RvwiServiceBase class holds the functionality for writing converting the state and characteristics information using the InstanceInfo class.

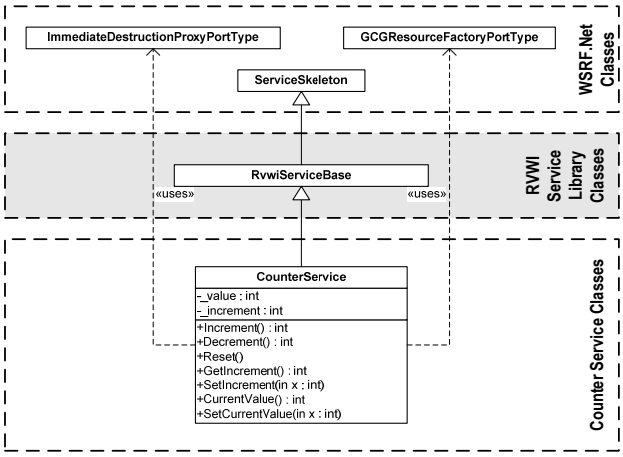

Diagram 5.1: Class Structure of the Counter Service

The CounterService class defines the Service itself. It is a simple class to test the recording of state and characteristic information to the database. The Counter-Service has one state element, the current value, and one characteristic, the amount used to increment and decrement the current value. The Counter Service also had methods to adjust the counter value, change the counter's increment and retrieve the current value of the counter.

5.2. Testing Process

Testing of the RVWI Counter Service required two clients: (i) the Generator Client and (ii) the Counter Client. The Generator Client was used to create ten Instances of the Counter Service. This was done by simply contacting the Counter Service and using the creation methods provided to it by the GCGResource-FactoryPortType class.

Aside from creating and preparing Instances of the Counter Service, the Generator Client tests the RVWI Documentation Module's ability to modify the WSDL. If the Instances appear in the database but not in the WSDL, then the Module has failed. It needed to be seen if the WSDL was still usable after being modified by the Module.

After creating an Instance of the Counter Service, the Generator Client would again contact the Counter Service (via the EPR given during the Instance's creation) and set the Increment and Value of the Instance. By default, all created Instances of the Counter Service have a Value of zero and Increment by one.

To test the discovery of web services based on state and characteristics, the Generator Client used methods in the Counter Service to modify the Instance, thus giving the set of Instances some diversity. Table 5.1 shows the parameters of all ten Instances generated by the Generator Client.

To check that the Generator Client finished properly, the WSDL of the Counter Service was requested and checked to see if all ten Instances were accounted for and that their Increment and Value parameters were set according to Table 5.1.

Instance #	1	2	3	4	5	6	7	8	9	10
Increment	1	2	3	4	5	6	7	8	9	10
Value	0	5	10	15	20	25	30	35	40	45

Table 5.1: Parameters of the Counter Service Instances

The Counter Client tested the use of the modified WSDL in selecting a proper Instance of the Counter Service. The Counter Client operated by getting the WSDL to the Counter Service, examine the Instance Information using XPath and selected an EPR to an Instance that matched the following requirements:

1. The Instance must increment by 5 or higher, and
2. The Instance must have a value of 40 or less.

Given the set of requirements, it is expected that only Instances 5 through to and including 8 will be selected.

If the Counter Client found multiple matches (as it should, given the above requirements) it listed the Instances and allowed the end user to select an Instance. After an Instance was selected, the Instance was incremented ten times and the value of the Counter Instance displayed on the terminal. To ensure that the Instance was being updated properly, the WSDL of the Counter Service was requested and checked to see if the Instance Information matched to the outcome of the Counter Client.

5.3. Testing Environment

The Counter Service, Generator Client and Counter Client were run on the single machine. The machine was an Asus Notebook with (i) Intel Centrino Duo Processor – a dual core processor, each core running at 1.6 GHz, (ii) 1 Gigabyte of memory, (iii) Windows XP Operating System with Service Pack 2, and (iv) .Net Framework 2.0

The Counter Service was hosted in the IIS version 5 Application Service (which comes bundled with Windows XP) and the Generator Client and Counter Client were run within Visual Studio .Net 2005.

6. Results

The first step of the testing was to run the Counter Generator and then capture the resulting Modified WSDL. Figure 6.1 shows a simplified version of the Modified WSDL with the first InstanceInfo element shown in bold (❶ in Figure 6.1). The other nine Instances have been collapsed to save space.

Note that this Modified WSDL was first edited by WSRF.Net. WSRF.Net makes use of the new WSDL Version 2 (World Wide Web Consortium, 2007) features currently in proposal stages with the World Wide Web Consortium (W3C) (W3C, 2007). The definitions of all the Service methods are not included and have been kept in separate WSDL files (❷ in 6.1). This feature is called 'importing' and is beyond the scope of this document.

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions xmlns:rvwi="http://www.deakin.edu.au/brock/rvwi"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <rvwi:InstanceDocumentation>
    <rvwi:Instances>
      <rvwi:InstanceInfo EPR="a674b589-3b57-4acb-9f42-719490658413">
        <rvwi:State>
          <rvwi:DescriptionElement Name="Value">
            <rvwi:Value xsi:type="s:int">20</rvwi:Value>
          </rvwi:DescriptionElement>
        </rvwi:State>
        <rvwi:Characteristics>
          <rvwi:DescriptionElement Name="CounterIncrement">
            <rvwi:Value xsi:type="s:int">5</rvwi:Value>
          </rvwi:DescriptionElement>
        </rvwi:Characteristics>
      </rvwi:InstanceInfo>                                          ❶
      <rvwi:InstanceInfo EPR="70bb71a3-b530-4f4c-a142-56f4e6a1681b">...
      <rvwi:InstanceInfo EPR="8297ff38-cdfb-4d03-ad8b-4d352951bab4">...
      <rvwi:InstanceInfo EPR="e4ccb0bb-9050-41f4-aef6-50be4e0b1fea">...
      <rvwi:InstanceInfo EPR="e570ed07-5dc0-474a-a6a7-655e059e96a2">...
      <rvwi:InstanceInfo EPR="4a15609d-17d6-41e7-aa62-a846870e3235">...
      <rvwi:InstanceInfo EPR="03cbe355-d1ea-40e7-b785-f33e234cd9cd">...
      <rvwi:InstanceInfo EPR="2d35e1b7-1308-432e-bef1-919bc427bdec">...
      <rvwi:InstanceInfo EPR="6a264d79-6ec9-4db4-a9a5-3dae0f4d93d5">...
      <rvwi:InstanceInfo EPR="247a4f50-4ee4-4e3e-b18a-dd7a7a7c5689">...
    </rvwi:Instances>
  </rvwi:InstanceDocumentation>
  <wsdl:import namespace=
      "http://www.deakin.edu.au/brock/rvwi/rwvi-counter/bindings"   ❷
      location="./RvwiCounterBinding.wsdl" />
  <wsdl:types />
  <wsdl:service name="RvwiCounterService">
    <wsdl:port name="RvwiCounterBinding"
          binding="i0:RvwiCounterBinding">...
    <wsdl:port name="RvwiCounterBinding1"
          binding="i0:RvwiCounterBinding">...
  </wsdl:service>
</wsdl:definitions>
```

Figure 6.1: Modified WSDL Before using Counter Client

After creating the ten Instances, the Counter Client was executed to find a desired Instance listing in the WSDL. For testing, an Instance that incremented at five or more and had a value of less than forty was desired. The Counter Client retrieved the WSDL and ran an XPath Query to find matching InstanceInfo elements. Figure 6.2 shows the Counter Client finding four such Instances and asking the end user to pick one.

```
Please select an instance of your choice.  To quit, enter 'Q'
0) a674b589-3b57-4acb-9f42-719490658413 increments at 5 and has a value of 20
1) 70bb71a3-b530-4f4c-a142-56f4e6a1681b increments at 6 and has a value of 25
2) 8297ff38-cdfb-4d03-ad8b-4d352951bab4 increments at 7 and has a value of 30 ❶
3) e4ccb0bb-9050-41f4-aef6-50be4e0b1fea increments at 8 and has a value of 35
Please Select an Instance =>
```

Figure 6.2: Starting the Counter Client.

As expected in Section 5.2, only Instances five through to and including eight were listed. The next step to the testing was to make use of an Instance. Instance 7, (❶ in Figure 6.2) was chosen and the Counter Client made use of the Instance by incrementing it ten times. Figure 6.3 shows the output after selecting an Instance.

Before incrementing the Instance, the Counter Client displayed the current value of the Instance (❶ in Figure 6.3). After each increment, the Counter Client displayed the value of the counter (❷ in 6.3). The purpose of this output is to see during execution that the Counter Service is behaving properly and that its state is safely being recorded and retrieved. When finished, the Counter Client displays the final value of the Instance (❸ in 6.3). It is this value that was looked for when the WSDL was requested again.

```
Please Select an Instance => 2
The current value of the counter is 30 ❶
Counter has been incremented to 37
Counter has been incremented to 44
Counter has been incremented to 51
Counter has been incremented to 58
Counter has been incremented to 65                                  ❷
Counter has been incremented to 72
Counter has been incremented to 79
Counter has been incremented to 86
Counter has been incremented to 93
Counter has been incremented to 100
The value of the counter is now 100 ❸
To quit the client, enter, 'Q'.  Otherwise, press enter
```

Figure 6.3: Counter Client Incrementing an Instance.

Note that after incrementing the Instance ten times, it now falls outside the requirements. To see if the Instance was omitted the WSDL was checked, the Counter Client was ran gain. This time, it was expected only Instances 5, 6 and 8 would be listed. Figure 6.4 shows the output from the Counter Client and also shows that the Instance has been omitted.

```
Please select an instance of your choice.  To quit, enter 'Q'
0) a674b589-3b57-4acb-9f42-719490658413 increments at 5 and has a value of 20
1) 70bb71a3-b530-4f4c-a142-56f4e6a1681b increments at 6 and has a value of 25
2) e4ccb0bb-9050-41f4-aef6-50be4e0b1fea increments at 8 and has a value of 35
Please Select an Instance =>
```

Figure 6.4: Rerunning the Counter Client.

Once the Counter Client was finished, the WSDL was requested again to see if the related InstanceInfo element was updated. Figure 6.5 shows the affected InstanceInfo element from the newer WSDL document. The 'Value' XML element (❶ in 6.5) has the value of the Counter Instance that was modified by the Counter Client. To conserve space, the other Instances and the rest of the WSDL was omitted.

```
<rvwi:InstanceDocumentation xmlns=
      "http://www.deakin.edu.au/brock/rvwi">
  <rvwi:Instances>
    <rvwi:InstanceInfo EPR=
          "8297ff38-cdfb-4d03-ad8b-4d352951bab4">
      <rvwi:State>
        <rvwi:DescriptionElement Name="Value">
          <rvwi:Value xsi:type="xsd:int">100</Value>                ❶
        </rvwi:DescriptionElement>
      </rvwi:State>
      <rvwi:Characteristics>
        <rvwi:DescriptionElement Name="CounterIncrement">
          <rvwi:Value xsi:type="xsd:int">7</Value>
        </rvwi:DescriptionElement>
      </rvwi:Characteristics>
    </rvwi:InstanceInfo>
  </rvwi:Instances>
</rvwi:InstanceDocumentation>
```

Figure 6.5: Instance Element After Using Counter Client

The testing demonstrated that web service selection can be simplified by including Instances in the WSDL. Previously, once a web service was discovered, the web service had to be accessed so its state could be learned.

Furthermore, while there was a guarantee that the discovered web service could meet the functional requirements of the Consumer, there was no guarantee of effectiveness. The web service maybe able to do the processing the Consumer wants but may take hours while another service can do it in seconds.

7. Conclusions and Further Work

Recently, web services started being adopted for use in software as a service thanks to the innovation of WSRF. Furthermore, recent work has looked at the means for listing the QoS and State of services in both UDDI and WSDL. Unfortunately, solutions lean toward services being of single instantiation or kept to the two step approach in getting service state.

RVWI addresses the problem in an innovative way by bundling state (multiple Instances) of a web service into its WSDL. Thus RVWI made it possible to find both the functions and current activity of the service easily accessible. The inclusion of characteristics allows clients to 'quantify' the resources that make up the desired web services. RVWI achieved this goal by making further use of web service Instances, snapshots of web service activity. Instances were used as they contain information directly from the related web service.

RVWI has presented a simplified means of selecting web services. Previously, discovery on aspects other than functionality were only possible by contacting additional web services or by manually querying the desired web services. RVWI granted to web services the ability to add state and characteristics information in to the WSDL which allows for one step queries.

RVWI is a significant step forward with state and characteristics now accessible through the WSDL. This allows the state of a web service to be learnt early in the selection process of web services. The significance is further enforced by the inclusion of characteristics in the WSDL. Questions about web services, such as "How big can a file be?" and "How many records can I put in this database?" can now be answered by looking at the characteristics of a web service. Previously, the only questions a Consumer can ask were along the lines of "Can this database hold records?" and "Can I read and write to this file?"

Another major innovation of RVWI is the simplification of web service brokering to the point of XML processing. Instead of holding data on web services or contacting additional services, XML queries can be run on the readily available WSDL. Also, RVWI supports Services having multiple Instances where previous solutions leaned toward single Instance (state) Services.

In order for RVWI to be fully deployable, it needs to be expanded to support the exposure of resource which can undergo state changes between Consumer requests. Also, RVWI needs some code generators added to simplify the authoring of Services to RVWI. Finally, performance testing needs to be carried out to see how much overhead is incurred when using RVWI. Performance testing is also required on RVWI Service. It needs to be determined how much slower RVWI Services perform against WSRF.Net Service.

References

Apache Software Foundation (2007) Apache Tomcat. Web Site. Accessed 25 July 2007. <http://tomcat.apache.org/>

Douglas Bryan, et al. (2002) Universal Discovery, Description, Integration. Specification. Updated 19 July 2002. <http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm>

Erik Christensen, et al. (2001) Web Services Description Language (WSDL) Version 1.1. Technical Report. Updated 15 March 2001. <http://www.w3.org/TR/wsdl>

Microsoft Corporation (2007) Microsoft ASP.Net. Accessed 25 July 2007. <http://www.asp.net/>

K. Czajkowski, et al. (2004a) *The WS-Resource Framework*. 5 March 2004. <http://www.globus.org/wsrf/specs/ws-wsrf.pdf>

Karl Czajkowski, et al. (2004b) Modeling Stateful Resources with Web Services. Report. Updated 3 May 2004. <http://www.globus.org/wsrf/>

Andrea D'Ambrogio (2006) 'A Model-driven WSDL Extension for Describing the QoS of Web Services'. *IEEE International Conference on Web Services (ICWS'06)*, 789-796.

M. Humphery and Glenn Wasson (2005) Architectural Foundations of WSRF.Net. *Internation Journal of Web Services Research,* vol. 2**,** p. 83-97.

Microsoft Corporation (2007) Microsoft .Net Homepage. Web Site. Accessed 17 July 2007. <http://www.microsoft.com/net/default.aspx>

Shuping Ran (2003) A model for web services discovery with QoS. *SIGecom Exch.,* 4**,** 1-10.

Brett Sinclair, Andrzej Goscinski and Robert Dew (2005) Enhancing UDDI for Grid Service Discovery by Using Dynamic Parameters. *ICCSA 2005.* Heidelberg, Springer Berlin.

W3C (2007) World Wide Web Consortium (W3C). Web Site. Accessed 21 June 2007. <www.w3c.org>

Glenn Wasson (2006) WSRF.Net. <http://www.cs.virginia.edu/~gsw2c/wsrf.net.html>

World Wide Web Consortium (2007) Web Services Description Language (WSDL) Version 2.0. W3C Proposal Document. Updated 23 May 2007. Accessed 21 June 2007. <http://www.w3.org/TR/wsdl20-primer/>