A Simplified Approach to Web Service Development

Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn

School of Computer Science
University of Adelaide
South Australia 5005, Australia
{pmk,paulc,andrew}@cs.adelaide.edu.au

Abstract

Most languages used for developing web services and clients exhibit properties which make calling remote functions across a network a non-trivial task. The type systems used by object oriented languages have many incompatibilities with those required for service interfaces, and the complicated tasks of generating proxy objects and WSDL service definitions mean that a lot of effort is required to create a service, in comparison with defining classes and functions to be used locally. We discuss the problems with existing systems and propose a new model for web services development based on an implementation of XSLT that we are currently developing. This provides a number of features useful for distributed applications such as automatic fault tolerance and load balancing, as well as a seamless mechanism for exposing and accessing web services.

1 Introduction

The concept of *service oriented architectures* (SOA) has been around for a long time (White 1976). It is based around the model of a series of computers on a network each providing a set of *services* which can be accessed by *clients*. A service consists of a set of named *operations*, each of which takes a number of parameters and returns a result. An operation is similar to a function defined in a programming language, except that it can be called over a network. The mechanism used to invoke these operations is typically referred to as a Remote Procedure Call (RPC) (Soares 1992). Numerous protocols and programming interfaces have been proposed over the years for this model, including SUN RPC, CORBA, RMI, DCOM, XML-RPC, and, most recently, web services. While most of the concepts behind web services are far from new, it is the standardisation of the technologies and widespread adoption now being seen today that makes them significant.

The term *web services* encapsulates a number of related standards which can be used together as a platform for developing distributed applications through the integration of new and existing systems. Web Service Description Language (WSDL) (World Wide Web Consortium (W3C) 2001) provides a standard way of describing a service, including the operations it provides, the data types and parameters of those operations, as well as binding information indicating which protocols and network addresses can be used to access the service. XML Schema (World Wide Web Consortium (W3C) 2004) is used by WSDL to define the data types that the message parameters in a service call must conform to. Simple Object Access Protocol (SOAP) (World Wide Web Consortium (W3C) 2003) is used for encoding messages that are sent and received by service operations. SOAP is most commonly used over HTTP, but supports other protocols as well, allowing applications to choose other mechanisms for communication that may provide additional benefits such as performance or security. Universal Description, Discovery, and Integration (UDDI) (Ariba Inc., IBM Corp. & Microsoft Corp. 2000) is a standard for service directories that can be queried by clients to find a service matching a specific criteria. Like many standards designed for interoperability these days, all of these specifications are based on XML, in order to provide a well-defined syntax for encoding and exchanging data between heterogeneous systems.

The concept of SOA is one type of *distributed programming* model, in which applications are built from a series of components residing on different computers. In a SOA, an application is divided into multiple parts, each of which assumes responsibility for part of the overall processing. Some parts may provide access to databases, or implement business logic, while others may provide features such as monitoring, logging, and administration. The way in which an application is divided up in this manner is similar to shared libraries or class libraries used in other programming systems. This approach allows complex systems to be divided up into simpler parts which can be developed by different people or teams, and for some components to be reused by other applications. This latter benefit is particularly common in SOA, where the loose coupling of the model lends itself readily to services which provide specific functionality that is not necessarily targeted at any particular application or user. These services can then be used by any application which needs the relevant functionality.

Another benefit that can be gained from using the SOA programming model is parallelism. With a large number of machines available, all providing a particular service, computation-intensive processing can be performed by splitting the work up and assigning it to different machines. With the use of asynchronous operation calls, a central "master" node can issue calls to the service on each machine on the network, each of which performs some part of the overall computation. This is one of the reasons why SOA has become popular recently in the area of grid computing (Foster, Kesselman, Nick & Tuecke 2002). The ability to develop applications in a distributed manner and harness the power of large numbers of resources is a major benefit of the SOA approach.

Despite the benefits of SOA, developing and util-

ising services is a less-than-straightforward task in many programming environments. Most programming languages in use today were designed before services became popular, and thus have numerous limitations which make the development of distributed applications awkward and complicated. Support for remote procedure calls is rarely built directly into the language, and thus it is often necessary to use alternative mechanisms to invoke service operations. Other aspects of traditional programming languages such as type systems and handling of network errors also make distributed programming difficult, as discussed further in the next section. Of course, many distributed applications can and have been built in these languages; but the process of doing so, in our view, is not as straightforward as it perhaps could be.

In this paper we describe a different approach to developing distributed, service-based applications which solves a lot of the problems associated with using traditional languages. We are currently in the process of implementing our ideas in GridXSLT, a distributed, parallel implementation of XSLT that is currently under development. It is our view that for grid computing and SOAs to really succeed, they need to be made easy and natural to use, by abstracting away the details of the underlying service definition and communication mechanisms from developers.

2 Existing approaches to service development

2.1 Compiled, object-oriented languages

The majority of web services development today is done using languages such as Java, C#, and C++. These are based on the object-oriented paradigm, where each object has a set of *methods*, and one or more *fields*, or *member variables*. Function calls occur by invoking a method on a particular object. In addition to primitive field types such as integers and floats, *pointers*, or *references*, to other objects are allowed. The in-memory state of a running program thus consists of a graph of objects with references to each other. These languages generally need to be compiled into either machine code or virtual machine bytecode before being executed, and this compilation plays a significant role in the way the development process occurs.

Remote procedure calls in object oriented languages are typically implemented using *proxy*, or *stub* objects (Birrell & Nelson 1984). These objects implement a defined interface that is known to the client and the server, containing methods corresponding to the operations provided by the service. In order to make a remote procedure call, the program calls a method on the proxy object. The way that the method is implemented is that it encodes the parameters into the format required for transmission across the network, sends the request to the server, and then blocks. Once the response has been received, it decodes the message into the return type of the method and returns from the call. To the code that made the call, it appears just like a normal method call.

Despite the fact that remote procedure calls appear the same as local calls when using proxy objects, a major drawback is the need to generate the proxy code in the first place. This can add complexity to the build process, and often requires effort on the part of the programmer in order to set up. Because the code must be compiled, it is necessary to generate the proxy classes prior to compilation of the code that uses them. In order to generate a proxy for a web service, it is necessary to use a tool which takes the WSDL file as input, and produces a series of source or compiled files, such as classes in Java. Every time a new service is to be used by a program, these files must be generated, and if the service definition changes, the files must be re-generated again. This is in contrast to class libraries which simply require the updated library file, such as a JAR archive or DLL file, to be placed in an appropriate directory.

Another disadvantage for the programming interface for services provided by object-oriented languages is the mismatch between the type systems used by the language and that used for web services. All web services define their input/output parameter types using XML Schema, which contains a number of features that are incompatible or at least awkward to implement in an object oriented language (Meijer, Schulte & Bierman 2003). These include:

Multiple element occurrences Most languages do not allow multiple fields of a class to share the same name, as this could lead to ambiguity when one of the fields is referenced. XML Schema, however, allows an element to contain multiple child elements with the same name, possibly interleaved with other elements. It is possible to model adjacent element occurrences as an array in an object-oriented language, however it is not possible to directly model cases such as a field "A" followed by a field "B" followed by another field "A". This would generally lead to ambiguity when the "A" field was referenced as it would be unclear which one was meant. Some tools for translating from XML Schema to other languages will rename the second instance of the field, but this is a sub-optimal solution.

Anonymous inner types In XML Schema, it is possible to specify the type of an element in-line, rather than as a separate named type definition. This is not widely supported in other languages. It is possible to work around this when translating from XML schema by creating named type definitions with automatically generated names.

Mixed content XML Schema also allows an element to contain a mixture of structured content, in the form of elements and attributes, and arbitrary text that can appear between the elements. In this case, there is no direct correspondence between the text and a field in a class definition, and so this information is generally lost when translating to other languages, or must be represented in an awkward way, such by having the class containing an array corresponding to the actual fields and text nodes present within the element.

Object-oriented languages also possess a number of features in their type systems that cannot be translated into XML, or cannot be meaningfully sent across a network connection:

Non-serializable objects There are certain types of objects that cannot be translated meaningfully into a stream of bytes. These include threads, socket connections, open file handles, user interface objects, and references to shared memory segments. The reason they cannot be serialized is that they do not just represent data, but also state that is shared with other parts of the system. Some RPC systems support these objects by transmitting a reference, which the server then uses to send messages back to the client when it needs to access the object (Tanenbaum & van Renesse 1988). For example, RMI does this with objects that implement the `Remote` interface in Java. However, this mechanism is not

supported by web services; only serializable objects can be sent as part of a SOAP call.

Pointers A running instance of a program may contain arbitrary graphs of objects connected together via pointers. When each object only has a single pointer to it, and is encapsulated as part of another object, then the graph represents a tree, and can be transmitted without loss of meaning. However, if there are multiple pointers to an object, then the serialized data sent to a web service operation must contain multiple copies of that object. This means that the identity of the object is lost, and it appears as two separate objects when the stream is decoded. Java's serialization mechanism handles this by encoding pointers, however this is not supported by serialization mechanisms which produce XML (Hericko, Juric, Rozman, Beloglavec & Zivkovic 2003), which are used for web service calls. SOAP does provide a form of support for pointers by allowing elements in a message body to act as references to other elements, but this approach is not supported by the XML schema type system used in service definitions.

Properties Some languages, notably C#, allow an object to contain *properties*, which appear to a caller as member fields, but are actually implemented in code. If a property called `amount` is defined on object `A`, then the code `A.amount = 12;` will cause the `amount` property to be set to that 12. Unlike a normal field though, the value doesn't get set directly; instead, a setter method is called, which can perform any arbitrary processing. A similar mechanism is supported for getter methods. When an XML schema type is generated for a class containing properties, only the fields will be stored, and the properties will be excluded. Thus, when a client-side proxy is generated from this definition, even if the proxy is generated in C#, the properties will not be present.

The most common way to generate a WSDL definition for a class defined in one of these languages is to use a tool which parses the source code to locate the functions and types, which it then generates operations and XML schema definitions for. Many developers take this approach to avoid writing WSDL by hand. However, the incompatibilities described above can result in errors when generating schema types for built-in classes, especially if those classes have not been carefully written to avoid using these features.

For this reason, other developers prefer to first define their service interfaces in WSDL, and then subsequently implement their service in compliance with the interface (Shohoud 2002). Some consider this a superior approach as it emphasises the importance of the interface over the implementation, and is more likely to produce types that are clearly defined and usable by other languages. But hand-coding WSDL files is a rather tedious process due to the (arguably unnecessary) complexity of WSDL, in particular the large amount of code that needs to be written, so this approach comes at a high price. The same issue also applies to dynamically typed scripting languages, described in the next section.

2.2 Scripting languages

Another popular approach to web service development is to use a scripting language. These are interpreted languages which do not require a separate compilation step in order to execute them. They are generally designed for lightweight programming tasks, as they often exhibit lower performance compared to compiled languages. This tradeoff is acceptable in many situations, however, as the ease of programming and quicker development cycle can make working with these languages easier. Examples of scripting languages such as these include Perl, Python, PHP, and Javascript.

One of the things that makes scripting languages so appealing for writing server-side code is the ease with which it can be made available to clients. Developing a service in most Java environments is a relatively complicated process, involving the editing of several different configuration files, generation of proxy classes, compilation of source files, packaging into archives, and using administration tools to deploy the archive to an application server. Using Perl or PHP to write a dynamic web page, on the other hand, involves simply copying the file onto a web server. In fact, most development of dynamic web sites using these languages is done by editing the files in-place on a test server, and as soon as the file is saved in an editor it can be accessed from a web browser without any extra steps. This is also a desirable approach for developing web services.

Using a scripting language to write a web service client tends to be much simpler than a compiled language. Because there is no compilation step necessary, and the languages do not perform static type checking on a program, it is possible to write code that creates a proxy object and calls methods on it, without the implementation of that proxy object being created until runtime. When the program executes, the SOAP implementation of the language can take a service definition and dynamically create a proxy object with the required methods based on the operations found in that definition. In many implementations, the WSDL definition of the service can easily be obtained at runtime and used for construction of the proxy object.

The dynamic typing of such languages can, however, be a drawback. For large programs, it is desirable to know at the start of execution whether or not there are errors in the program, rather than wait to find out about them when the relevant code comes to execute. Compiled languages like those described previously will catch errors like calls to non-existent methods and passing of incorrect types as parameters; these things will only occur at runtime in a scripting language. This issue, however, is one that is common to the choice of programming language regardless of whether the task is web services development or some other type of program, and scripting languages have nonetheless seen widespread popularity.

Another disadvantage that comes from the lack of static typing in scripting languages is the problems involved with automated generation of WSDL files. These languages do not permit user-defined functions to specify their parameter types or result type, but this information is required by WSDL. If a client written in Java wants to access a service written in Python, then the client needs to know the types it should be sending. For WSDL definitions to be written for services implemented in a scripting language, it is therefore necessary to manually create the definitions.

2.3 Web service composition languages

Another class of languages, often referred to as *web service composition* languages, are also used for building services. The most prominent of these is Business Process Execution Language (BPEL) (Andrews, Curbera, Dholakia, Goland, Klein, Leymann, Liu, Roller, Smith, Thatte, Trickovic & Weerawarana

2003), which is targeted specifically at writing services that form part of an overall pattern of interaction with other services. It provides constructs for invoking operations on remote services and storing the results in variables that can be then passed to other operations. BPEL programs are themselves exposed as web services and are invoked upon requests from clients.

A major drawback of BPEL is that while it does contain some limited control constructs, the expressiveness of the language is fairly restricted compared to many other languages. For example, conditional and loop statements are supported, but there is no way to create user-defined functions or complex data structures. It is thus difficult to write application logic in BPEL; it must instead be implemented in services provided by other languages, and simply called from the BPEL program. Additionally, the verbose, XML-based syntax of BPEL requires several times the amount of code to be written for a given task compared to most other languages.

Other service composition languages such as SCUFL (Oinn, Addis, Ferris, Marvin, Greenwood, Goble, Wipat, Li & Carver 2004) are based on the workflow model, where the programmer creates a directed graph in which the nodes correspond to operations and the edges indicate the flow of data between the nodes. While this model readily lends itself to parallelism, writing programs directly for this model is often a difficult process due to the fact that it is fairly low-level, and a very different programming model to what most programmers are used to.

Similar work to what we propose in Section 4 has been done on extending XQuery to support web service development and consumption (Onose & Simeon 2004). This comes close to achieving many of the goals we are aiming for; however, XQuery is a less powerful language than XSLT (Kay 2005) and the XQuery implementation developed in this work lacks the parallelism and distributed execution features that we are investigating as part of our research, although these are outside the scope of this paper. Another XML-based language, XL (Florescu, Grnhagen & Kossmann 2003), also shares many of our goals, however our approach is to innovate at the implementation level rather than proposing a new language.

2.4 Summary

The object-oriented and scripting languages described above, which represent the most common approaches to distributed services programming today, were originally designed for stand-alone environments. Network support and distributed processing were subsequently added as extras, rather being part of the core language. While they are useful for developing a wide range of sequential, non-distributed applications, the act of connecting together different services in these languages is not a straightforward, natural process. Each language has gone part way to supporting the streamlining of the process, such as the dynamic proxy generation present in scripting languages, or the automated generation of service interface definitions from statically typed, object-oriented languages, but still, additional effort is required to create a service beyond merely writing the relevant functions to be exposed.

More recently, a number of languages specifically targeted at web service composition have been proposed, which address some of these issues, but in general lack many useful features and provide a comparatively restricted programming model. As such, they are mainly appropriate only for simple compositions of services written in other languages which provide more complex functionality. The work done on XQuery and XL provides a richer programming model for working with services; our work is most closely related to these projects.

In this paper, we propose an implementation of the XSLT language. Our implementation leverages an already existing, powerful language, and extends it with functionality that makes it appropriate for web service development and composition. While the semantics of the language make certain compromises on features compared to some mainstream languages, it is still possible to implement complex application logic in the language in addition to gaining features that make web service development easier and more seamless than in other languages.

3 Desired language features for web service development

Based on our analysis of existing systems for developing web services and clients, we have identified a set of requirements which we feel are necessary to meet our goals of providing an easy to use development environment. It is our opinion that a system based on these features will make the process of building clients and services easier and more natural. These requirements are:

- The language should support static typing, so that WSDL definitions can be automatically and seamlessly generated from function definitions in the source code. The type system used should fit within the web services model, in order to avoid the problems described in Section 2.1. A programmer should never have to run into a situation where they define a data type, use it extensively throughout their code, and discover later when they try to expose a web service operation using the type, that it uses some feature that is not supported by the serialization mechanisms and has to be changed, along with all the code that uses it.

- All data should be serializable to XML so that it can be sent as part of a SOAP call. This means no support for pointers or objects representing system state. The programmer should never have to worry about whether or not a data structure they have created is going to cause problems when they want to send it to another node on the network. This is true *network transparency* - the network is there, but you can't see it. Working with local data is the same as working with remote data.

- A rapid development cycle should be supported, being as easy to use as Perl and PHP scripts are for building dynamic web sites. A developer should be able to save their source code in an editor, and have the service immediately accessible to clients.

- The WSDL definition of a service should be automatically and seamlessly generated from the code of the service itself without any intervention from the programmer. It should not be necessary to manually create WSDL files, or even to explicitly run generation tools. A service developer should not even have to know what WSDL is.

- There should be no difference between local and remote function calls, either in terms of writing functions or calling them. Whenever a function is added to a program, it should be implicitly and automatically callable as a web service operation. Any web service should be accessible by

importing it into the program by specifying its URL, after which all of the operations can be called as if they were locally-defined functions. There should be no need for proxy classes or serialization code to be generated at compile time.

- To facilitate error recovery and load balancing, all functions should be side-effect free. A service may not maintain state; whenever an operation is called, it performs some computation based on its parameters, and returns a result. There should be no impact on any external entity as a result of calling an operation. This can be used to allow for retries in the event of network errors, or for service operation calls to be load balanced across a set of machines providing identical services.

These requirements place certain restrictions on the types of applications that can be written, particularly the lack of support for pointers, and the assumption that all functions are side-effect free. It is important to recognise that this model is not appropriate for all types of applications, as in some cases these features are necessary. While these restrictions may seem problematic for certain applications, we have taken the view that in order to gain some of the benefits, such as automatic error recovery, load balancing, and network transparency, it is appropriate to make these trade-offs. For applications which can be developed within these restrictions, we believe our approach offers a significantly more intuitive model of distributed programming.

It is likely that in many situations developers may find it useful to integrate programs developed within this model with those written in the other languages described above; this can be easily done as all of the integration can be achieved using the web services standards. This situation is similar to that of other specialised languages, for example SQL, which is well suited to database queries, but almost always needs to be used in conjunction with another language in order to form a complete application. This model is thus appropriate for *web service composition* (Khalaf, Mukhi & Weerawarana 2003), which involves composing a set of web services implemented in other languages to produce a high-level program.

Our approach of using a restricted, functional programming model is similar to that used for distributed query processing (Kossmann 2000). Clustered and distributed database systems partition their data across multiple sites, and execute queries by performing some portion of the processing on each machine, usually close to the data being accessed. To the client, this distribution is completely hidden, and the request is submitted in SQL or some other high-level query language. A database programmer never has to concern themselves with how to structure their code for concurrent execution, or deal with threads and message passing. Sophisticated compilation and optimisation techniques are used within the database engine itself to execute the query in parallel. Database query languages are relatively restricted in their programming model, but still very widely used as part of other applications. The large body of existing work in this area demonstrates the viability of using a high-level approach to distributed programming.

We see our work as augmenting existing technologies for developing web services, rather than being a replacement for existing approaches. A major benefit of SOA is the ability to connect together heterogeneous systems and enable interaction between them in a platform and language independent manner. It is likely that in the majority of usage scenarios for our system, our language implementation will be used to compose together functionality provided by existing services written in other languages. Our work is targeted at providing both facilities for web services composition, and for implementing application logic. The degree to which the language is used for each of these tasks is up to the programmer; we provide support for both types of development so that the choice is available.

4 Overview of GridXSLT

GridXSLT is an implementation of the XSLT programming language (World Wide Web Consortium (W3C) 2005) that we are currently developing, which is designed to meet the requirements listed above. XSLT, or Extensible Stylesheet Language Transformations, is a pure functional language designed for dealing with XML data. It is commonly used for transforming data from one format to another, typically for presentation formatting or integration of different business systems. Existing implementations of XSLT are designed for processing relatively small amounts of data, and do not contain support for web services. Our implementation is aimed at large-scale, distributed data processing of the sort becoming common in the area of grid computing.

XSLT possesses a number of features which make it suitable for achieving our goals. All variables and parameters consist of simple or complex types that can be represented in XML. There is no support for pointers, and it is not an object oriented language, although it is still possible to define arbitrarily complex data structures. A complex data structure is one that consists of multiple member fields, which are either simple types such as integers or strings, or other complex data structures. While in many languages these are supported using pointers, in XSLT the data structure model is restricted to containment only. This is similar to using nested structs in C and avoiding the use of pointers. Most types of data structures, including trees and lists, are supported by this model. All data types are defined using XML schema, which is the same type system used by WSDL - this means that any custom data type declared in the program can be included directly as part of the service interface without any incompatibilities.

Our implementation supports automatic parallelisation of XSLT code. As a functional language, all functions are side-effect free, meaning that there is no concept of state in a running program which can change due to a function call. This means that the order in which a set of functions are called within the code does not effect the result of the program. In an expression containing calls to several different functions, each of these can be executed concurrently, potentially across multiple machines. Additionally, variables are single-assignment only, so once a variable has been given a value it is guaranteed to retain that value for the remainder of its lifetime. This is another important feature required for automatic parallelisation. The way in which this parallelism and distributed execution operates is not discussed in this paper; the reader is referred to (Kelly, Coddington & Wendelborn 2005) for a more detailed overview.

The GridXSLT execution engine runs programs using an interpreter, so there is no need for programs to be compiled before they are run. Unlike the interpreted scripting languages described earlier, however, the language supports static typing. Before program execution begins, the source code is translated by the interpreter into an intermediate form, and part of this process is to type check all variable assignments and procedure calls. This checking is based on the function signatures for locally defined functions, and the

WSDL definitions of all services that the program uses. Because the service definitions contain all of the type information for parameters and results, it is possible to ensure that the calls are using variables of the correct type. The names of all functions referenced in the code are also checked to ensure that they correspond to either a locally available function, or a web service operation declared in a WSDL file.

The implementation of calls to web services happens directly within the interpreter, instead of using proxy objects. Functions in XSLT are grouped into namespaces, in a similar manner to which classes are grouped into packages in Java. When a program uses a service, it associates a namespace with that service, specifying the location of the WSDL definition. When a function call occurs during program execution, the interpreter inspects the namespace of the function, and if it has been mapped to a service definition, then the call will be redirected to the service instead of a local function. This is described further in the next section

4.1 Access to other web services

XSLT does not provide direct support for web services as part of the language specification. However, it does allow for implementors to provide *extension functions*, which are additional functions that are outside of the specifications. An extension function is specific to a particular implementation, so code that uses these is not portable to other implementations of the language, unless they also provide the same extension.

Namespaces are used to group functions together, and when an extension function is provided, it has a particular namespace associated with it. To call the function, it is necessary to associate a prefix with the namespace using the standard mechanisms provided by XML. When the function is called, this prefix is placed at the start of the function name. For example, to call the function `getPrice()` in the namespace `urn:store`, the following code could be used:

```
<xsl:value-of
  xmlns:st=''urn:store''
  select=''st:getPrice('shoes')''/>
```

In our implementation we use this mechanism to make web service operations accessible as extension functions. When a namespace prefix is declared with a URL that begins with `wsdl:`, we treat this as a namespace containing functions corresponding to the operations provided by the service. In order to call a function on a web service, all that is necessary is to associate a namespace prefix with a URL containing the WSDL definition of that service, and then all function calls in that namespace will result in requests that are sent to the service. For example:

```
<xsl:value-of
  xmlns:st=''wsdl:http://store.com/api?WSDL''
  select=''st:getPrice('shoes')''/>
```

Normally the namespace prefix would be declared once at the top of the program, rather than individual statements as in the examples here. Apart from this declaration, use of a service appears exactly like local function calls. There is no need for any extra steps such as generation of proxy classes, as the method calls and serialization are all handled internally within the execution engine.

Because of the support for static type checking, and the ability to check that all references to web service operations are correct, runtime errors relating to invalid use of service operations that can occur in other scripting languages are not possible in our implementation. Any such errors are reported before program execution begins.

4.2 Exposing programs as web services

A web service provides a set of operations which are accessible to clients; each operation corresponds to a function that is implemented by the service. When an XSLT program containing user-defined functions is exposed as a web service, the service contains one operation for each function. The parameters and return value of each function correspond to the input and output messages of the operation.

The GridXSLT execution engine acts as a web server and exposes all programs within its document root as services. No special action is required on the part of the developer to expose their program as a service other than placing the source file in an appropriate directory on the server. When the engine receives a request for the file, it executes the operation specified by the client. All decoding of parameters and encoding of the result value is handled by the engine itself, and there is no need for serialisation code to be generated separately for each complex type used by the service.

A WSDL definition of a service is automatically generated by the engine when the client submits a request for the filename with "?WSDL" appended to the end of the URL. All information about the operations and parameter types can be determined by analysing the source file, and this is combined with information about the address of the server to generate the binding information which tells the client how to access the service. Because the type system used by XSLT is identical to that of WSDL, there is no chance of user-defined types not being expressible in the service definition. While most other languages require the programmer to be careful when defining data types, in order to ensure that features of the type system not supported by web services are avoided, all schema types defined in an XSLT program are guaranteed to be serializable, and thus usable as part of a service interface.

A WSDL file consists of five different sections: types, messages, port types, bindings, and services. The *types* section contains XML schema definitions for all complex types used by operation parameters or results. A namespace must be declared for the schema, and each type is given a name, which is referenced from other parts of the document. The *messages* section contains a separate input and output message for each operation provided by the service. A message consists of one or more named parts, each of which references a built-in XML schema type, or one of the complex types defined in the WSDL file.

The service definition also contains one or more *port types*. These are basically the same as interfaces, and contain a set of operations. Each operation in the port type corresponds to a function implemented by the program, and has an input and output message. The messages defined previously are referenced by name within each of the operations. Despite the fact that the message parts are declared in order previously, it is necessary to declare the parameter order again in each operation element.

Next comes the *bindings* section. This declares the messaging protocol to be used for each port type. Every single operation must be repeated here, this time giving specific information about how that operation should be called. The information required here depends on the messaging protocol being used; for the common case where the SOAP protocol is used, the type of encoding for the message body is specified.

The last section is the *services* section, which lists the port types that are provided by the service and associates the previously declared bindings with them. It is this element which determines the actual overall set of operations that are provided. It is possible to have multiple port types, and multiple bindings for each port type, although in the common case there is only one of each.

This is a considerably more complex method of defining service interfaces than that provided by most other RPC systems. The example given in Section 5 requires no less than 52 lines of code to represent a service with a single operation, for which the type definitions and function signature (i.e. without the implementation) requires 2 lines of XSLT. Since WSDL is an rather verbose language, it is beneficial to provide a mechanism to express interface definitions in a more concise syntax, which is what we have done in our implementation, as demonstrated in the example. As mentioned in Section 2, some web service development environments provide the ability to automatically generate WSDL definitions from the source code, however the issue of type system compatibility can often cause problems. With XSLT there is a 1:1 mapping between the types expressible in the language and those exposed via WSDL, as both languages use XML Schema.

4.3 Fault tolerance and load balancing

In distributed programming systems, fault tolerance is an important issue. When a client makes a call to a remote service, it is possible that a network error may cause the request to fail. This error can be due to a variety of reasons, including the remote host crashing, the network being disconnected, or a high load on either the network or one of the machines causing the request to time out. When developing a distributed application it is necessary to deal with these situations in some way.

The simplest way to handle an error is to simply stop execution of the program. In the case where the client itself is actually a web service that has been called from yet another host, this equates to returning from the call with an error, rather than the expected result value. If the client is an application invoked directly by a user, then the program simply exits with an error message.

In most cases, however, it is desirable to try to recover from the error. The most obvious way is to retry the call until it completes, and then continue execution. This is done if the error was a transient one, such as a timeout, which stands a chance of succeeding on a second attempt. However, if the error is permanent, such as a response indicating that the service does not exist, or the parameter types are incorrect, then future retries will likely fail, so execution is instead aborted.

Another error handling approach supported by the engine is to retry the call on a different host. This is possible when the WSDL definition of the service includes multiple bindings, each referencing a separate host. It is possible for the provider of a service to set up a number of machines which all provide identical implementations of the same service, and include each of these as alternative bindings in the service definition. If these are available, then subsequent retry requests are sent to machines for which communication errors have not occurred, in preference to those which have.

This approach of multiple bindings is also used for load balancing. If a service is to be accessed multiple times, then every time a request is made, a different host will be used. When a number of different requests are made by concurrently executing portions of the client program, this enables distributed parallelism, potentially resulting in a shorter overall execution time for the program. It also ensures that where multiple machines are available, they each get a fair share of the load, rather than one machine having to handle all the requests.

The fault tolerance mechanisms supported by GridXSLT rely on the assumption that the web services being used are side-effect free, in line with the semantics of the XSLT language. It is possible for a situation to arise where a request is received by the server, and the operation is performed, but a subsequent network error causes the response to be lost. The client will interpret this as an error and retry the call. If the service maintains state, then this could cause undesirable results, such as a credit card being charged twice. State is also a problem for load balancing, because if each service maintains its own state, then each request could see a different service state depending on where it is sent.

Our initial implementation is restricted to web services which are *stateless*, which means that the only effect that calling an operation has is to return a result value, since there is no state to be modified, and multiple calls to an operation will return the same result each time. Obviously, not all web services operate on this model, and these are not supported. As discussed in Section 3, this restricts the scope of applications that can be developed within our model. However, we feel that for a restricted class of applications, i.e. those which do not rely on state, the ability to transparently perform error recovery and load balancing in this manner is an attractive feature. One area we may explore in the future is how stateful services can be supported within this model.

The fault tolerance and load balancing features of our system are only in the early development stage, so no results yet are available. The ideas we have presented here will be further developed and described in more detail in future publications.

4.4 Condensed language syntax

An important feature we are implementing in our engine is support for an alternative language syntax for XSLT. The default syntax defined by the language specification is rather verbose, as everything is represented by XML elements. Like other XML-based languages, considerably more code is required to express the same things that can be expressed much more concisely and succinctly in other languages. For example, the code to call a template with a series of parameters looks like this:

```
<xsl:call-template name=''foo''>
  <xsl:with-param name=''a'' select=''12''/>
  <xsl:with-param name=''b'' select=''3''/>
</xsl:call-template>
```

We are developing an alternative syntax, XSLiTe, which contains all of the same statements as XSLT but allows them to be expressed in a manner which is much more straightforward to work with. Programmers familiar with languages such as C, Python, and Java will likely find this a more natural fit. An example of the above template call represented in this syntax is as follows:

```
foo(a=12,b=3);
```

We will not go into further details here, as a more detailed example is given in (Kelly et al. 2005). Our execution engine is capable of handling programs written in either this condensed syntax, or the standard XML-based syntax of XSLT, and can translate

between the two language syntaxes. The only difference between the two is the *syntax*, not the *semantics* - the available language constructs are the same in both, and the same program written in either syntax will produce the same results.

5 Example

In order to demonstrate the use of GridXSLT for developing a web service, we give a simple example of a matrix multiplication service. A single operation, `mmul()`, is provided, which takes two 3x3 matrices as parameters and returns the result of multiplying them together. Each matrix is represented as an XML structure containing a series of `<row>` elements, each of which contains `<col>` elements with the values of the numbers in each cell.

The code for this service is shown in Figure 1. The first line defines a data structure called `matrix` which is used as the type for both parameters of the function as well as the return value. This type definition uses the condensed XSLiTe syntax to represent an XML schema definition. The `mmul()` function is then defined next, specifying two parameters, `a` and `b`. The code to multiply the matrix consists of two nested loops, within which the value for each cell in the result matrix is computed based on the contents of the two supplied matrices. The `%row` and `%col` expressions specify element creation; the result of the `sum()` function in the innermost loop is placed within a `<col>` element each time it is called, and a ¡row¿ element is created for each iteration of the outer loop containing the column elements for that row.

To expose this program as a web service, it is simply necessary to place the file within a directory under the document root of a running GridXSLT engine. A client may then submit requests to the server for the `mmul()` function, or obtain the service definition by appending "?WSDL" to the URL. In the latter case, the WSDL definition is dynamically created and returned to the client, as shown in Figure 2. This contains the XML schema type declaration for the `matrix` type, generated from the source code of the service. It also includes an operation for the `mmul()` function, as well as input and output messages for the operation with the appropriate type associations. The binding and service elements are also generated for the service, based on the address of the web server.

Two things are worth noting about the generated WSDL file. Firstly, the type definition is taken verbatim from the source, with no mapping between different type systems. Although the source code uses the more condensed syntax, it is expressing the same thing as the complex type defined in the types section. Secondly, the difference in the amount of code required to express the type and operations of the web service is more than an order of magnitude greater than in the XSLiTe syntax in which the program is written. While this could be automatically generated for a statically typed language, it would have to be written by hand for a dynamically typed scripting languages, cancelling out the rapid development benefits for which those languages are often chosen.

6 Conclusion

In this paper we have presented an analysis of existing web service development environments and pointed out the fundamental limitations in the popular languages used today which make distributed programming a difficult task. We have made the case for an alternative programming model that is more suited to distributed systems, based on the goal of obtaining network transparency, by hiding low-level details from the programmer. In particular, the capabilities of fault tolerance and load balancing that we have described ease the task of dealing with multiple hosts in a grid environment, particularly when large scale applications are involved that require good performance and reliability.

We have discussed the ways in which our XSLT implementation provides solutions to these problems, by providing seamless mechanisms to consume and expose web services. As we are currently in the process of implementing this design, we have not yet been able to perform a full evaluation of this approach, but we feel that it is likely to result in a superior development experience for the programmer, due to the hiding of low-level details and making distributed programming a natural part of the language.

We see this as part of a continuation of the evolution of programming languages that has been going on for several decades. Initially when some functionality is made available, programmers are expected to deal with it explicitly, and then higher level tools, APIs, and languages appear which abstract the details away. It used to be that applications were required to manage their disk storage on a block-by-block basis, and then file systems were invented so that the operating system could take care of it. In the early days of the PC platform, programmers had to interface directly with hardware such as sound cards and graphics cards, and then device drivers and operating system APIs came along which hid the details and provided a common interface. A similar evolution has occurred with the advent of Java, and programmers can now write platform-independent programs without having to worry about low-level operating system details.

Our work is an application of these ideas to the problem of developing distributed applications using web services. In particular, we are concerned with hiding the details of how services are exposed and accessed, how parallel programs are divided up and distributed across a grid of machines, and how data can be represented in a way that can be seamlessly encoded and safely transmitted across a network. Working in a higher level programming environment like the one we have proposed here will bring the benefits of abstraction to the field of distributed web service programming.

Further information about this project is available at `http://gridxslt.sourceforge.net/`.

References

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I. & Weerawarana, S. (2003), 'Business Process Execution Language for Web Services version 1.1', http://ifr.sap.com/bpel4ws/.

Ariba Inc., IBM Corp. & Microsoft Corp. (2000), 'Universal description, discovery and integration (UDDI) technical white paper'. http://www.uddi.org/.

Birrell, A. D. & Nelson, B. J. (1984), 'Implementing remote procedure calls', *ACM Transactions on Computer Systems (TOCS)* **2**(1), 39–59.

Florescu, D., Grnhagen, A. & Kossmann, D. (2003), 'XL: An XML programming language for web service specification and composition', *Computer Networks Journal (Special Issue on Semantic Web)* **42**(5).

```
type matrix { { int col[3]; } row[3]; };

matrix mmul(matrix $a, matrix $b) {
  for-each $i in (1 to 3) %row
    for-each $k in (1 to 3) %col
      sum(for $j in (1 to 3) return $a/row[$i]/col[$j] * $b/row[$j]/*[$k]);
}
```

Figure 1: Implementation of the matrix multiplication service

```
<?xml version="1.0"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:tns="urn:matrix"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             targetNamespace="urn:matrix">
  <types>
    <schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="urn:matrix">
      <complexType name="matrix">
        <sequence>
          <element name="row" minOccurs="3" maxOccurs="3">
            <complexType>
              <sequence>
                <element name="col" minOccurs="3" maxOccurs="3" type="xsd:int"/>
              </sequence>
            </complexType>
          </element>
        </sequence>
      </complexType>
    </schema>
  </types>
  <message name="mmulResponse">
    <part name="mmulReturn" type="tns:matrix"/>
  </message>
  <message name="mmulRequest">
    <part name="a" type="tns:matrix"/>
    <part name="b" type="tns:matrix"/>
  </message>
  <portType name="MatrixPortType">
    <operation name="mmul" parameterOrder="a b">
      <input message="tns:mmulRequest" name="mmulRequest"/>
      <output message="tns:mmulResponse" name="mmulResponse"/>
    </operation>
  </portType>
  <binding name="MatrixBinding" type="tns:MatrixPortType">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="mmul">
      <soap:operation soapAction=""/>
      <input name="mmulRequest">
        <soap:body use="literal"/>
      </input>
      <output name="mmulResponse">
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="MatrixService">
    <port binding="tns:MatrixBinding" name="MatrixPort">
      <soap:address location="http://localhost:8080/matrix"/>
    </port>
  </service>
</definitions>
```

Figure 2: WSDL file for the matrix multiplication service

Foster, I., Kesselman, C., Nick, J. M. & Tuecke, S. (2002), 'The physiology of the grid: An open grid services architecture for distributed systems integration', Open Grid Service Infrastructure WG, Global Grid Forum.

Hericko, M., Juric, M. B., Rozman, I., Beloglavec, S. & Zivkovic, A. (2003), 'Object serialization analysis and comparison in java and .NET', *ACM SIGPLAN Notices* **38**(8), 44–54.

Kay, M. (2005), Comparing XSLT and XQuery, *in* 'XTech 2005: XML, the Web and beyond', Amsterdam, The Netherlands.

Kelly, P. M., Coddington, P. D. & Wendelborn, A. L. (2005), Distributed, parallel web service orchestration using XSLT, *in* '1st IEEE International Conference on e-Science and Grid Computing', Melbourne, Australia.

Khalaf, R., Mukhi, N. & Weerawarana, S. (2003), Service-oriented composition in BPEL4WS, *in* 'World Wide Web 2003 Conference, Web Services Track', Budapest, Hungary.

Kossmann, D. (2000), 'The state of the art in distributed query processing', *ACM Computing Surveys (CSUR)* **32**(4), 422–469.

Meijer, E., Schulte, W. & Bierman, G. (2003), Programming with circles, triangles and rectangles, *in* 'Proceedings of XML 2003'.

Oinn, T., Addis, M., Ferris, J., Marvin, D., Greenwood, M., Goble, C., Wipat, A., Li, P. & Carver, T. (2004), Delivering web service coordination capability to users, *in* 'Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters', ACM Press, New York, NY, USA, pp. 438–439. http://taverna.sf.net.

Onose, N. & Simeon, J. (2004), XQuery at your web service, *in* 'WWW '04: Proceedings of the 13th international conference on World Wide Web', ACM Press, New York, NY, USA, pp. 603–611.

Shohoud, Y. (2002), 'Place XML message design ahead of schema planning to improve web service interoperability', *MSDN Magazine* **17**(12).

Soares, P. G. (1992), On remote procedure call, *in* 'CASCON '92: Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research', IBM Press, pp. 215–267.

Tanenbaum, A. S. & van Renesse, R. (1988), A critique of the remote procedure call paradigm, *in* R. Speth, ed., 'Proceedings of the EUTECO 88 Conference', Elsevier Science Publishers B. V. (North-Holland), Vienna, Austria, pp. 775–783.

White, J. E. (1976), A high-level framework for network-based resource sharing, *in* 'Proc. National Computer Conference'.

World Wide Web Consortium (W3C) (2001), 'Web services description language (WSDL) 1.1'. http://www.w3.org/TR/wsdl.

World Wide Web Consortium (W3C) (2003), 'Simple object access protocol (SOAP) version 1.2'. http://www.w3.org/TR/soap.

World Wide Web Consortium (W3C) (2004), 'XML Schema part 0: Primer second edition', W3C Recommendation. http://www.w3.org/TR/xmlschema-0/.

World Wide Web Consortium (W3C) (2005), 'XSL transformations (XSLT) version 2.0', W3C Working Draft. http://www.w3.org/TR/xslt20/.