JWS: A Flexible Web Service

Andrew Cho, Paresh Deva, Ewan Tempero
Department of Computer Science
University of Auckland
Auckland, New Zealand
ewan@cs.auckland.ac.nz

Abstract

Web services have been proposed as means to provide more convenient access to computation services. An issue that still must be dealt with is what to do if there is no web service with the desired functionality. Deploying a new web service requires expertise in the relevant technologies as well as access to a web services server. In this paper we present the Java Web Service, a web service that allows the provision of almost arbitrary functionality by means of uploading the functionality as a *plug-in* at run-time. Plug-ins can also be combined through a simple scripting mechanism.

Keywords: Flexible computation service, Web services, Distributed systems.

1 Introduction

The ability to access computation across a network or to distribute computation around a network has been a goal of distributed systems research for many years. Each new generation of distributed systems technology removes one more barrier to providing such an ability. The most recent step has been the introduction of *web services*, which provide network access to to software systems in an *interoperable* manner (Booth, D. et al. 2004), meaning that use of the systems is language and platform independent.

An issue that still remains with web services is that the services they offer are under the control of whoever controls the servers. If the required functionality does not exist as a web service, then there are few options. Those that need new functionality can create it themselves but if they want to make it available to others then they face starting and managing their own server. For those who would rather not take this route, we propose creating a web service that is *flexible* — the functionality it provides can be added to at runtime by any user without affecting existing users. In this paper, we describe the *Java Web Service* (JWS), a web service that can be configured at runtime to allow almost arbitrary functionality.

Web services are claimed to provide several advantages over other distributed computing technologies, including:

- Interoperability — Requests and responses are encoded in neutral formats, such as XML. This means there is no requirement that client and server have to agree in advance on programming language, operating system, or hardware, in order to communication. For example, a Java web service client running on OS X could communicate with a web service provider implemented in C# running on windows.
- Reuse — The interoperability aspect of web services allows an existing module to be wrapped by a web service implementation and have its functionality made available to clients on different software platforms. This avoids the need to duplicate functionality for each platform.
- Open standards — Web services use open non-proprietary standards such as XML and HTTP. This ensures no one company has control over web services. This has helped the popularity of web services.

We wish to retain these advantages as much as possible, while at the same time adding:

- Flexibility — Provide a general computation service by supporting any functionality. This flexibility should not come at the cost of down time. Many clients may be using the service through the Internet at any time and so restarting the service with new functionality is not practical.
- Security — The service should not cause harm to the server that it is running on, such as the deletion of files.

The basic approach we take is to develop a web service with a *plug-in* architecture, in a manner similar to applications such as Eclipse (Gamma & Beck 2004). New functionality can be uploaded to the web services server at run-time, and plug-ins can be combined through a simple scripting mechanism.

The remainder of the paper is organised as follows. The next section provides the background on web services necessary for understanding the rest of the paper. Section 3 describes JWS, with details on the design and implementation in section 4. Section 5 gives an example use of JWS. In section 6 we give an evaluation of the success of JWS, and then discuss related work in section 7. Finally, we give our conclusions and discuss possible future work in section 8.

2 Web Services

Web services are software systems that can be used over a network in an interoperable manner (Booth, D. et al. 2004). They can be used by any program that can process eXtensible Markup Language (XML) and access a network. The simplest form of web services involves two parties: a web service *provider* (server) and a web service *requester* (client). The web service provider makes available some functionality that it performs on behalf of the web service clients. The client, of which there could be many, consumes the

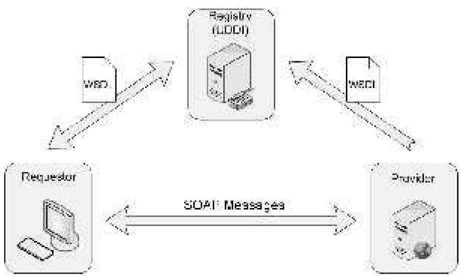

Figure 1: Web Service Architecture Diagram

web service and calls the provider's functionality. The web services concept can also involve a third party, a web services *registry*. Clients may browse this registry for web service providers with specific functionality and bind to them dynamically. Figure 1 illustrates this case.

The provider is responsible for publishing its functionality by implementing a service interface. Providers are described through a service description file. This file is written in the *Web Service Description Language* (WSDL), which is a language for describing web services based on XML. The service description file describes the web service's functionality, such as the available methods, arguments and return types, and the service's network location, and is published to the registry (Booth, D. et al. 2004).

A web service client may not know exactly which web service provider it will interact with. Therefore the client must "discover" a web service provider. Web services can be discovered either manually or automatically. In manual discovery, the client's developer may hard-code the client to use a particular web service. In automatic discovery, web service providers publish their WSDL documents to a registry service, such as Universal Description Discovery and Integration (UDDI). Clients then use this registry service to search for suitable web service providers, obtain their WSDL documents, and bind to them dynamically.

After binding, the requester and provider communicate by sending *Simple Object Access Protocol* (SOAP) messages to each other. SOAP encodes the parameters and results of a service invocation into XML. These messages are transported over a communication protocol such as HTTP, SMTP, FTP, IIOP, or proprietary protocols.

3 The Java Web Service

The JWS provides a flexible computation service that clients can invoke over a network. It provides flexibility in two ways: the JWS can compile and execute almost any Java source code, and JWS has the ability to have its functionality extended by uploading *plug-ins*. These two features combine to allow the JWS to support almost arbitrary functionality.

Figure 2 provides an overview of the JWS. The JWS has a minimalist core of a web service interface, script manager, and plug-in manager. The web service interface receives requests from web service clients and forwards them to the appropriate component. The plug-in manager is responsible for loading, configuring, and invoking plug-ins. The script manager invokes scripts. Plug-ins and scripts are described in more detail in sections 3.2 and 3.3 respectively.

By themselves, these core components provide very little functionality for clients. All of the useful functionality are implemented as plug-ins. Plug-ins provide the first mechanism for the JWS's flexibility in that new functionality can be uploaded to the JWS as a plug-in. They are integrated dynamically into the JWS while it is running and this integration causes no down time.

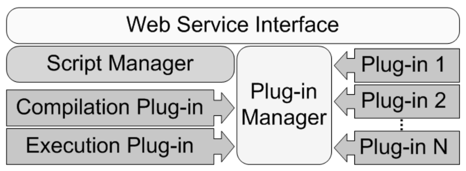

Figure 2: Overview of the Java Web Service

```
<?xml version="1.0" encoding="UTF-8"?>
<ExecutionResult ExitValue="0"
         PluginName="ExecutionPlugin">
  <ExecutionSuccessful>
    true
  </ExecutionSuccessful>
  <ExitValue>
    0
  </ExitValue>
  <Filename>
    TestExecSampleClass
  </Filename>
  <StandardOutput>
    Hello World!
  </StandardOutput>
</ExecutionResult>
```

Figure 3: Example document describing the result from the `execute` method applied to `TestExecSampleClass`
.

The compilation and execution plug-ins are responsible for compiling and executing Java source code respectively. They provide the second mechanism for the JWS's flexibility in that they can be used to execute almost any Java source code. These plug-ins are bundled with the JWS and are always available.

3.1 JWS Service Methods

The JWS contains several different service methods that can be invoked by its clients. These are:

`compile` This method takes one argument that is a string. The string represents the code that the client requires to be compiled. The compilation plug-in is called from this method and the result of this call is a string (an XML document) describing the result of the compilation that is sent back to the client.

`execute` This method takes one argument that is the name of a Java class file and calls the execution plug-in to execute this file. There is a form of this method that takes an integer as a second argument. This argument specifies the amount of the time that the execution is allowed to continue for before the execution process is killed. In the first form of this method, the time is defaults to ten seconds. As with `compile`, an XML document is returned describing the result of the execution. An example is given in figure 3.

`uploadPlugin` This is the method that installs a plug-in provided by the client. Its argument is an array of byte codes. This method converts the data into a file and saves this file in the plug-in repository directory (specified at start-up). A boolean is returned indicate whether or not the upload was successful.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin name="EchoPlugin"
        class="echoplugin.EchoPlugin">
  <pluginDescription>
    This plug-in is just for testing
  </pluginDescription>
  <method>
    <name>
      echo
    </name>
    <argument>
      java.lang.String
    </argument>
    <description>
      Echoes back the given String
    </description>
  </method>
</plugin>
```

Figure 4: An example of a plug-in manifest file

`getListOfPlugins` This method is used to send information to the client about what plug-ins currently exist in the plug-in directory. It returns an XML document containing the names and descriptions of all of plug-ins. It also contains the names, descriptions and arguments of all of the methods of the plug-ins.

`runScript` The runScript method is used to run a script submitted by the client. It accepts two arguments: script and input. The script is a string and is of the format specified in section 3.3. The input is a string that represents the argument to be used for the first plug-in as specified in the script. The Script Manager executes the script and returns an XML document describing the result of the execution.

3.2 Plug-ins

A plug-in is a small program that provides certain functionality. It can be uploaded to the JWS in order to add new functionality. The JWS's plug-in architecture resembles a simplified version of the Eclipse plug-in architecture (Gamma & Beck 2004). Eclipse is also an application that has a minimalist core of functionality that is extended by plug-ins. As with Eclipse, plug-ins for JWS are packaged within a Java Archive (JAR) file. This archive contains the plug-in's class files and an XML manifest file. The manifest tells the JWS the name of the plug-in, the class to instantiate, and the methods the plug-in has to offer. An example of a manifest file is shown in Figure 4.

For the example in figure 4, the name of the plug-in is "EchoPlugin" and its implementation class is "echoPlugin.EchoPlugin." It only defines one method, "echo", which accepts a `java.lang.String` object as an argument.

The plug-in's implementation class must implement an interface specific to JWS called `IPlugin` so as to be recognised by the JWS as a plug-in. This interface defines two methods: `init()` and `shutdown()`. These methods are called just after instantiation and before removal respectively. They give the plug-in the opportunity to configure and release resources.

3.3 Scripts

Clients submit scripts to the JWS that define which plug-ins should be invoked. Ideally the JWS would publish any methods that plug-ins make available in its WSDL document. However if a new plug-in was uploaded, then the WSDL document would need to change and this would require the web service's clients to re-acquire the latest version of the WSDL and possibly be recompiled. The JWS would also need to be recompiled and re-deployed to Apache Axis, because the JWS's implementation class would need to implement these new methods. As we want to avoid down time, this approach is impractical.

```
NumberPlugin multiplyBy3
EchoPlugin echo
```

Figure 5: An example of a simple plug-in script

```
IF FAIL NumberPlugin multiplyBy3
  EXIT
ELSE
  EchoPlugin echo
ENDIF
```

Figure 6: An example of a plug-in script with conditional behaviour

Scripts allow the WSDL document to remain the same while still enabling new plug-ins to be invoked. As mentioned above, the JWS defines a method, `runScript(Stringscript,Stringinput)`, that executes the given script using the given input, and returns the result of executing the script. A script is essentially a list of plug-ins in the order they are to be executed. Scripts operate in a similar fashion to the UNIX pipe and filter interaction paradigm with the output of one plug-in being passed to the input of the next plug-in.

An example of a script is shown in Figure 5. This simple script uses two plug-ins. The plug-in named `NumberPlugin` has a method called `multiplyBy3` that converts the script's input to a number, multiplies it by three, and returns the result. If the input into this script were "100" then the output after the first line of the script would be "300." This result is passed onto the plug-in named `EchoPlugin`. The echo method of this plug-in concatenates its input string with itself. If this method were executed with an input of "300" the result would be "300300." Therefore the result of executing the example script with an input of "100" is "300300."

Scripts also have the ability to support conditional behaviour based upon whether a plug-in succeeds or not. Figure 6 shows an example of such a script. This script says: if the `NumberPlugin` fails then exit the script, otherwise use the `EchoPlugin`. The `NumberPlugin` will fail if the input into the script can not be converted into a number. Conditional behaviour allows the path of execution to change based upon the outcome of a plug-in.

Plug-ins can tell the JWS whether they have failed in their execution in two ways. Firstly, a plug-in can set an exit value state variable that the JWS will check. This exit value uses the convention that a value of zero means the plug-in was successful and a non-zero value means failure. The second method involves the plug-in returning an XML document. The root node of the document defines the attribute "ExitValue" that contains the exit value described above (similar to that shown in figure 3). The exit value contained within the document will override the exit value contained within the state variable.

The scripting language is very simple, providing

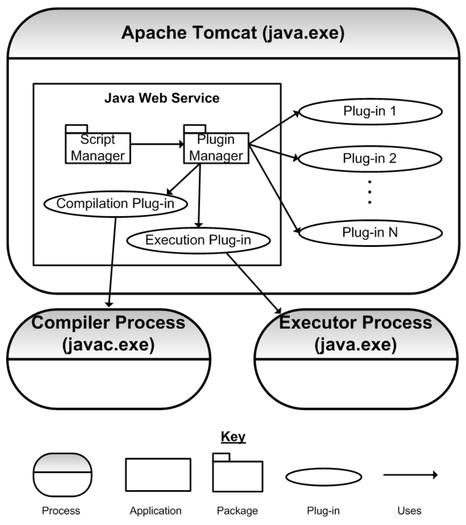

Figure 7: Architecture of the JWS

only the sequencing and conditional behaviour shown here, however we believe that this is sufficient for a wide range of applications.

4 Design and Implementation

4.1 Technologies

Java was chosen because the Java Virtual Machine (JVM) allows the same compiled source code to be run on all operating systems where the JVM is available. This allows development and deployment to become operating system independent.

The web server chosen was Apache Tomcat (Apache 2006*a*) due to its ease of deployment and local expertise. By itself, Apache Tomcat can not host web services because it does not include an implementation of SOAP. Apache Axis (Apache 2006*b*) provides this implementation and also includes several tools to aid web service development and deployment.

From the perspective of the JWS's clients, the technology choices have negligible impact. Web services are interoperable and the implementation details of the service, such as the programming language and choice of server, are hidden away from clients. The only impact of these choices is that plug-ins must be written in Java. Allowing plug-ins to be written in any language was outside the scope of this project but would further improve the flexibility of the JWS.

4.2 Architecture

Figure 7 shows the process structure of the JWS. All of the processes shown reside on the server that hosts the JWS. The main process running on the JWS server is the Apache Tomcat JVM. This process runs the Apache Axis web application (not shown), which in turn runs the JWS. The script manager processes incoming scripts and uses the plug-in manager to invoke plug-ins. The compilation and execution plug-ins are shown to be within the JWS application because they are always available.

To compile Java source code the compilation plug-in writes the source code to a `.java` file and saves it on the file system. Then the `javac` process is started, which compiles the `.java` file to a `.class` file. The execution plug-in then starts the `java` process to execute the `.class` file in its own JVM. The plug-in returns any output that the program sent to either standard output or standard error.

Unlike submitted Java source code, uploaded plug-ins are loaded and executed within the same JVM as the JWS. Submitted Java source code is executed in its own JVM because it can generate output by calling `System.out.println(...)`. This method returns output to the JVM's console and not to the caller. There is only one console per JVM and if multiple programs sent their output to the console it would be difficult to differentiate the output of one program from another. There is a performance cost for this decision as we discuss in section 6.

4.3 Security

The security concerns for the JWS are more prevalent than traditional web services. With traditional web services the service provider is not modified by the client. The JWS is not only modified by uploading plug-ins but arbitrary Java code can also be executed. Both uploaded plug-ins and submitted source code could attempt to cause harm to the server. This includes manipulating the file system (reading, modifying, or deleting files), starting new processes (starting a UNIX shell and executing commands), or opening a network connection allowing back-door access to the server.

To minimise the effect of malicious code harming the server, Java's security manager is enabled for both the Apache Tomcat JVM and the JVM in which submitted Java source code is executed. By default the security manager gives all code none of the permissions specified within the Java Security Architecture framework (Gong 2002). Java security policy files are used to grant certain permissions to certain Java classes. The compilation plug-in is allowed to write to the file system because the compilation process requires writing `.java` and `.class` files to the file system. The execution plug-in is allowed to read these files. To maximise security, all uploaded plug-ins are given no permissions. Uploaded plug-ins can not be trusted because they could be written by anyone.

Security policy files are incapable of preventing infinite loops. An infinite loop is a situation where a program executes continuously inside a loop and never reaches the condition that tells the program to exit the loop. To prevent infinite loops from wasting CPU time on the server, the JWS executes submitted Java code in its own process and kills its process if it goes beyond a certain time limit. This time limit can be modified to the client's demands. More expensive computational tasks will require a longer timeout. However uploaded plug-ins are executed in the same process as the JWS so this approach can not be used for plug-ins that are in an infinite loop. This and other security issues are discussed in section 6.

4.4 Communicating between plug-ins

To achieve complex behaviour, plug-ins may be chained together in a script. For example there may be the need to invoke the execution plug-in after using compilation plug-in. Coordination between plug-ins is difficult because plug-ins could potentially be developed by different developers. If the output of one plug-in does not match the input of the next plug-in then the communication breaks down and the operation will fail. This issue is further complicated by plug-ins that could potentially pass anything (text, files, images) to the next plug-in.

Ideally a plug-in will accept and return a string of text formatted in XML. XML is helpful because it can be used to represent any object and is machine readable, allowing it to be processed dynamically at runtime.

Although passing objects instead of XML would have improved performance (the task of transforming objects to XML and back to objects again would be avoided) it would be a less flexible solution. For a plug-in to pass an object to another plug-in, the receiving plug-in must have an implementation of that object. Therefore there would need to be an agreement between the two plug-ins at design time. This would severely limit the number of plug-ins a given plug-in could interact with.

Plug-ins still need to agree to what XML they expect to receive and produce. This is done by plug-ins providing a Document Type Definition (DTD) file of the XML they produce. This file is machine readable and is a standard way of defining the structure of an XML document.

5 JWS in Action

This section demonstrates how the flexibility of JWS would be used to build an application. The application we developed is called the Online Learning Application (OLA). This application aims to help beginner Java programmers learn the Java syntax. Students go to the OLA's website and are asked to perform Java programming exercises. An example exercise may be to write a for-loop that prints the numbers from one to ten. Students would then write a small piece of Java code that performs this task. The OLA would then compile and execute the student's submission, and compare the result of execution against an expected answer.

Using the OLA as a motivating example for the JWS raises some challenging issues. Novel solutions are required because these issues are not encountered with traditional web service implementations. These issues are outlined below:

- In addition to the JWS acting as an execution engine by executing Java code, it must also be able to compile Java code. Together, compilation and execution provide a flexible mechanism for performing computational tasks.
- Security concerns arise because the JWS could be asked to execute any code that students submit to it. This code could attempt to delete files from the server. Such behaviour must be prevented.
- Plug-ins must be able to be integrated into the JWS despite plug-ins and the JWS being developed by different people. Standardisation is required so that all plug-ins can be integrated into the JWS.
- It is desirable to have plug-ins that can communicate with each other. There must be some agreement on how to pass output from one plug-in to the input of the next plug-in. For example, the compilation plug-in should pass its output to the execution plug-in. The execution plug-in should be able to interpret the compilation plug-in's output and execute any compiled classes.
- Functionality should also be invoked conditionally. For example if the compilation plug-in fails in compiling Java source code, the execution plug-in should not be invoked.
- Some plug-ins require the allocation of machine-dependent resources. For example, the compilation plug-in must have access to a directory to compile its `.class` files to. This directory will be different for every server the plug-in is deployed on.

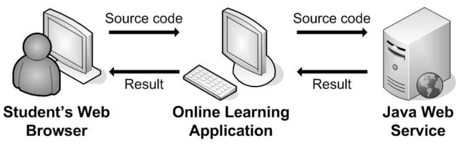

Figure 8: Overview of the Online Learning Application

```
for (int i = 1; i <= 10; i++)
    System.out.println(i);
```

Figure 9: Expected submission for a Java exercise

An overview of the OLA and its relationship with the JWS is shown in Figure 8. The OLA does not implement much functionality. It essentially passes students' submissions to the JWS for compilation and execution, and checks the result against an expected answer. This feedback is then passed back to the student. This demonstration illustrates how applications can be composed quickly by leveraging existing services. Any gaps in functionality can be filled by uploading a plug-in to the JWS.

Students access the OLA through their web browser and complete Java exercises. If the OLA asked the student to write the code that prints the numbers from one to ten, the student would be expected to write something similar to the code shown in Figure 9.

The OLA uses the script shown in Figure 10, with the student's code snippet as the input, to invoke the JWS.

First, the `WrapperPlugin`'s wrap method is invoked. Java requires code to be declared within a class. This plug-in wraps the student's submission with a class declaration and returns the result. For the above example, this plug-in would produce something similar to the class shown in Figure 11.

The name of class is made unique by appending a random number to the end. This minimises the chance that saving the `.java` and `.class` files to the file system will conflict with existing `.java` and `.class` files. These files need to be periodically deleted.

The rest of the script says the code should be compiled with the `CompilationPlugin`. If compilation is successful then `ExecutionPlugin` is used to execute the code. Any output from standard output or standard error is sent back to the OLA. The OLA checks this result against an expected answer and informs the student whether they were correct or not.

If the student's submission does not compile, the script will return the compiler's compilation error messages. These messages are difficult to understand and pose a significant obstacle for beginner programmers (Flowers et al. 2004, Lang 2002). It would be helpful if the OLA had the ability to convert these messages to a more understandable format. This gap in functionality can be filled by developing a plug-in.

```
WrapperPlugin wrap
IF SUCCESS CompilationPlugin compile
    ExecutionPlugin execute
ENDIF
```

Figure 10: Script used by the OLA to invoke the JWS

```
public class Test123456789
  public static
         void main(String[] args)
    for (int i = 1; i <= 10; i++)
      System.out.println(i);
```

Figure 11: Student's code inside a class declaration

```
WrapperPlugin wrap
IF SUCCESS CompilationPlugin compile
    ExecutionPlugin execute
ELSE
    CompilationErrorPlugin transform
ENDIF
```

Figure 12: Script with the CompilationErrorPlugin

`CompilationErrorPlugin` was developed to perform this task. It examines the compiler's error messages, attempts to match them against a catalogue of errors, and replaces them with a more descriptive message. The only change the OLA requires to use this new plug-in is to add an else condition to its script as shown in Figure 12.

This new script says if compilation does not succeed, then the compilation error plug-in should be used to transform the output of the compilation plug-in (compilation errors) into a more understandable format.

The potential for functionality of the OLA is not limited by the JWS. For example a user could decide to extend the OLA to accept a language other than Java by writing a plug-in that would provide this functionality, or if a user feels the `CompilationErrorPlugin` plug-in is insufficient, they can write a better one to be used in its place.

Similarly, the OLA does very simple checking that the student's submission is correct. This checking could be made more sophisticated, in which case it could be incorporated into the OLA application code itself, or made in to a plug-in and uploaded to the JWS server. This plug-in would then be available for anyone else to use.

The OLA requires user code to be executed on the web server, so security becomes an issue. The security policy (mentioned in section 4) for the user-submitted code is specified just before the code is actually run. This is allowable as security policies can be specified when a process is started up for execution, which occurs when the execution plug-in is used. This allows different policies to be specified for different code; so trusted users can use a less stringent policy, which grants more permissions.

For the OLA, a timeout facility is used to ensure that user submitted code that would never complete, such as infinite loops that would continually run but never get any closer to finishing, does not stall the server. After a set amount of time the code would time out (the 'timeout time'). This is implemented utilising the timeOutTime specified in the execution plug-in. For the OLA, the timeout time is set to ten seconds, as only beginner code is expected to be submitted, resulting in quick execution times.

6 Evaluation

The primary goals of this work were to develop a distributed computation system with functionality that is not determined just at deployment time, but allows changes to functionality with no down time, is interoperable, that is, was usable by any client regardless of the programming language, operating system, or hardware of the client, and is secure. By using web services and Java for interoperability, and a plug-in architecture for flexibility, JWS does, in theory, meet these goals.

We have provided evidence of the JWS's flexibility and interoperability in a more practical sense by developing OLA. We face that same problem of any new "enabler" technology. The technology by itself provides no directly observable functionality, so to really prove its usefulness requires devoting significant resources developing many different uses of it. In our case, we chose OLA because it exercises all the features provided by JWS in a single application, and we feel it is representative of a large class of applications of the kind that JWS is intended to support.

In particular, OLA has requirements that motivate the need for flexibility. Had OLA been developed as a standard web service, then its behaviour would be fixed at that time. If, for example, it is determined that the deployed presentation of compilation errors is not informative enough, the users must wait until a better presentation is produced and deployed. With our implementation of OLA users can provide their own plug-in to present such information.

JWS does, however, have its limitations.

- Computational tasks must be specified in Java, either by submitting Java source code or a plug-in. No support is provided for executing source code in other programming languages.
- The JWS is a designed to perform computational tasks and can not be used to display user interfaces and graphics.
- The Java security manager prevents code from causing harm to the server. Any tasks that require operations such as writing to the file system, the creation of new processes, or use of network connections will result in security violations. Currently there is no mechanism that grants permissions to uploaded plug-ins.

As mentioned earlier, the decision to execute submitted Java source code in its own process as opposed to loading it into the same process as JWS has a performance cost. Figure 13 gives an indication as to what this cost is. It shows the cost of performing a compilation, execution in a new process, or execution in the existing process. These timings were done using a Java class that executed an empty for-loop for approximately 16 milliseconds. The tests were run 100 times and the results averaged to minimise random variation. While this is a fairly simplistic test, it does provide the information we are interested in. From the test we can see that executing in a new process takes on average 140 milliseconds longer than executing in the same process. In the context of the various other performance costs associated with web services, we feel this extra time is acceptable.

There are a number of directions that JWS could go. As JWS provides a general computation service, it would be interesting to extend it to provide a distributed *parallel* computation service. To realise the benefits of parallel computing, this would require developing a a plug-in that allows one JWS server to communicate to other JWS servers. This plug-in would need to disseminate the computational task

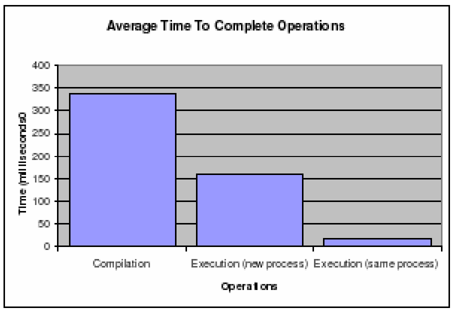

Figure 13: Performance of compilation and execution

amongst the other JWSs and recombine the results. This plug-in would require permission to open network connections. Ideally this plug-in should be bundled with the JWS in a similar manner as the compilation and execution plug-ins.

If parallel computation can be realised the JWS will resemble the Globus (Globus 2006) system. Globus provides the Globus Toolkit, which is used to create grid computing solutions and has become a middleware standard for a number of grid projects. Globus' web services grid resource allocation manager (WS-GRAM) provides dynamic deployment of web services.

The current form of JWS is as secure as the Java security manager is. The security policy we use rules out most security concerns when using JWS. As mentioned in section 4.3, security policies cannot prevent all malicious behaviour from occurring. An uploaded plug-in could execute an infinite loop and the JWS will be blocked while waiting for plug-in to finish. Since uploaded plug-ins are executed in the same process as the JWS itself, killing the process would also kill the JWS, Apache Tomcat, and any other web applications Apache Tomcat was hosting at the time. This issue could be addressed by executing uploaded plug-ins within their own process and killing their process if a timeout is exceeded. This is the same method for terminating infinite loops in submitted Java source code.

Like other web services, the JWS can benefit from the incorporation of the Web Services Security (WSS) policy (Open 2006). This policy aims to provide web services with message integrity and confidentiality. It specifies mechanisms for encrypting messages and authenticating clients and servers.

The WSS policy mentioned above provides a mechanism for authenticating clients but does not provide a mechanism for authenticating uploading plug-ins. It could be possible for an authenticated client to upload an untrustworthy plug-in. This could be addressed by using the JAR signing and verification tool, which is part of the Java Security Architecture (Gong 2002). This tool allows JAR files to be digitally signed so that the JWS can be sure about the origins of the plug-in. This is only part of solution because there must be some mechanism that ensures only trustworthy plug-ins are signed.

Currently the only resource that uploaded plug-ins are allowed to manipulate are the arguments that are passed to it. To allow more powerful plug-ins, such as the parallel computing plug-in, plug-ins must be able to request resources from the JWS. Granting resources is non-trivial because the JWS will not know what resources a plug-in requires until it is requested. In addition, the JWS and plug-ins have no prior agreement, therefore an arbitrary plug-in could request an arbitrary resource. The compilation and execution plug-ins are given a directory to write files to through the plug-ins' constructors. However these plug-ins are special case because they were developed as an integral part of the JWS. Foreign plug-ins can not be passed resources through their constructor because the JWS has no idea what resources are required.

The plug-in could specify the type of resource it requires in its manifest file. For example, the parallel computing plug-in could request a network connection by specifying the value "network connection" within a resources tag element and ideally the JWS will grant the plug-in the ability to create network connections. But there are still issues, such has how the JWS and plug-in developers know that the value "network connection" should be used as opposed to "socket connection," "network," or even "network permission".

Even if the JWS understood what the plug-in was requesting, it still may not know what object to return. For example, if the JWS was asked to provide a directory it may not be clear whether the JWS should return a string that represents a path to the directory, a URL, or a `java.io.File` object.

Prior agreement is required. The JWS could publish a list of resources and the values used to retrieve them as comments in its WSDL document or even a website. However this requires the plug-in developer to know the JWS it will be uploading the plug-in to at design time and modifying the plug-in's manifest file for each JWS server. This is inflexible and fragile if the JWS changes.

7 Related Work

Researchers have attempted to make web services more flexible as well. Most approaches relate to the dynamic composition of web services. This involves creating complex web services by dynamically integrating several simple web services together. In Sam et al.'s (Sam et al. 2006) implementation of this approach web service clients provide a specification of the web service they require to a registry service. If there are no web service providers that match the specification exactly, the registry returns the closest match. Then additional web services are used to fill the gaps in functionality and make the original web service match the user's specification more closely.

Sam et al. use the example of a Japanese tourist wanting to book a hotel in France via a web service. The registry returns a web service that allows the tourist to book French hotels however, it is in French and uses Euros as its currency, which is meaningless for the Japanese tourist. Intermediary web services are used to convert French to Japanese and Euros to Japanese Yen.

Although the hotel booking web service is made more flexible, it relies on these intermediary web services to already exist in the registry. Even if an intermediary web service does exist it may not fit the user's requirements exactly. In contrast, the JWS allows users to modify the JWS's functionality by uploading a plug-in. Since users can develop their functionality their requirements are matched exactly.

Another project looking at dynamic web services is the "Web Services Management Layer" (WSML) project (Verheecke et al. 2003). This project aims to remedy the problems caused by the traditional "Wrapper" approach to web services design, which is a static methodology, used in development environments such as Microsoft .NET. The WSML project creates a layer in between the web application and web service. This provides a layer of abstraction that can be used to allow "hot-swapping" of web services at run time. Different modules are present in this layer, controlling areas such as security and transaction management (Verheecke et al. 2003).

The idea of making the execution of program code a web service comes from a project on verifying dynamic reconfigurations of systems (Warren et al. 2006). This project extended OpenRec, a framework that comprises a reflective component model plus an open and extensible reconfiguration management infrastructure, by integrating it with the Alloy analyser (Jackson 2002). The integration of the existing OpenRec framework, written in Python, and Alloy, written in Java, was achieved by delivering Alloy as a web service.

8 Conclusions

The key objectives of this work were to create a service that is interoperable and flexible enough to allow new functionality to be added without down time and secure enough to prevent harmful behaviour from occurring at the server. JWS meets these goals through it being a web service and through its plug-in architecture and ability to compile and execute submitted Java code. Plug-ins can be submitted and will become available at run-time without affecting the server. A further advantage of its plug-in architecture is that plug-ins are available to any client of JWS, giving further opportunities for reuse.

Plug-ins are invoked through a simple scripting language. Scripts allow users to specify which plug-ins are invoked and the order of execution of these plug-ins. Simple conditional processing is provided. Whether such a simple language is sufficient for all envisioned uses of JWS is the subject of future work.

Security of the JWS is implemented through use of the Java Security Framework. The minimum number of permissions that still allow the JWS to function are given. No permissions are currently given to uploaded plug-ins, so no uploaded code can damage the server. The only security concern in the current implementation is that denial-of-service is possible through the submission of a long-running plug-in, something that could be addressed through alternative implementations. Indeed, the security policies of the current implementation probably reduce the flexibility of JWS more than is necessary, and providing a more flexible security model is also the subject of future work.

We are not the first to consider how to provide more flexibility in web services, but we have taken a different approach to other projects, namely providing a mechanism to give the user more control over the functionality provided by the web service. We believe our minimalistic design provides a convincing proof-of-concept that our approach is viable.

References

Apache (2006*a*), 'Apache Tomcat', Apache Software Foundation `http://tomcat.apache.org`.

Apache (2006*b*), 'Web services - Axis', Apache Software Foundation `http://ws.apache.org/axis`.

Booth, D. et al. (2004), 'Web services architecture: W3c working group note 11 february 2004', Retrieved 23rd August, 2006 from `http://www.w3.org/TR/ws-arch`.

Flowers, T., Carver, C. & Jackson, J. (2004), Empowering students and building confidence in novice programmers through gauntlet, *in* 'Proceedings of the 34th ASEE/IEEE Frontiers in Education Conference', pp. 10–13.

Gamma, E. & Beck, K. (2004), *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*, Addison-Wesley.

Globus (2006), 'Globus', The Globus Alliance `http://www.globus.org`.

Gong, L. (2002), 'Javatm 2 platform security architecture', Retrieved April 26, 2006 from `http://java.sun.com/j2se/1.5.0/docs/guide/security/spec/security-spec.doc.html`.

Jackson, D. (2002), 'Alloy: a lightweight object modelling notation', *ACM Trans. Softw. Eng. Methodol.* **11**(2), 256–290.

Lang, B. (2002), Teaching new programmers: A Java toolset as a student teaching aid, *in* 'Proceedings of the Inaugural Conference on the Principles and Practice of Programming', pp. 95–100.

Open, O. (2006), 'Oasis web services security (wss) tc', Retrieved August 27, 2006 from `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`.

Sam, Y., Boucelma, O. & Hacid, M.-S. (2006), Web services customization: a composition-based approach, *in* 'ICWE '06: Proceedings of the 6th international conference on Web engineering', ACM Press, New York, NY, USA, pp. 25–31.

Verheecke, B., Cibrán, M. A. & Jonckers, V. (2003), AOP for Dynamic Configuration and Management of Web Services, *in* 'The International Conference on Web Services - Europe', pp. 137–151.

Warren, I., Sun, J., Krishnamohan, S. & Weerasinghe, T. (2006), An automated formal approach to managing dynamic reconfiguration, *in* 'ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)', IEEE Computer Society, Washington, DC, USA, pp. 37–46.