Abstract

Dependability is a key factor in any software system due to the potential costs in both time and money a failure may cause. Given the complexity of Grid applications that rely on dependable Grid middleware, tools for the assessment of Grid middleware are highly desirable. Our past research, based around our Fault Injection Technology (FIT) framework and its implementation, WS-FIT, has demonstrated that Network Level Fault Injection can be a valuable tool in assessing the dependability of traditional Web Services. Here we apply our FIT framework to Globus Grid middleware using Grid-FIT, our new implementation of the FIT framework, to obtain middleware dependability assessment data. We conclude by demonstrating that Grid-FIT can be applied to Globus Grid systems to assess dependability as part of a fault removal mechanism and thus allow middleware dependability to be increased.

Keywords: Fault Injection, Dependability Assessment, Globus, Grid.

1. Introduction

The Globus Toolkit [1] is the front running Grid technology currently in use for large-scale scientific Grid applications. As such it requires close scrutiny in terms of its dependability [2] to foster trust and speed its introduction into other application domains. Globus Toolkit 4 is constructed from a number of base components and key to these is the Apache Axis SOAP stack which forms one of the message transports available, so dependability assessment of this would contribute greatly to the overall dependability of the toolkit.

Fault Injection [3] is a well-proven method for assessing the dependability of a system and has been used extensively in the domain of distributed systems, for example Marsden et al [4]. Recently there has been interest in applying fault injection to services. Network level fault injection has provided promising results in assessing the dependability of service based systems and middleware [5] and forms the basis of our FIT method.

Grid – Fault Injection Technology (Grid-FIT) is a dependability assessment method and tool for assessing Grid services by fault injection. Grid-FIT is derived from our earlier fault injector WS-FIT [5] which was targeted towards Java Web Services implemented using Apache Axis transport. Grid-FIT utilises a novel fault injection mechanism that allows network level fault injection to be used to give a level of control similar to Code Insertion fault injection whilst being less invasive [6].

This paper uses the Grid-FIT tool to systematically carry out a dependability assessment of Apache Axis 1.2 as used in the Globus Toolkit 4. We present our findings on case studies conducted to determine the integrity of data exchanged during service calls and their vulnerability to integrity attacks.

2. Grid Middleware

Globus Toolkit [1] is an open source software toolkit used for building Grid systems. A large part of Globus Toolkit is constructed around Web Services utilizing Apache Axis as the transport.

A Web Service is a service defined by a number of standards that can be used to provide interoperable data exchange and processing between heterogeneous machines and architectures. For the purposes of this paper, Web Services will be defined as being described by Web Service Definition Language (WSDL) [7] and implemented using SOAP and the Remote Procedure Call (RPC) model. Web Services are commonly used to provide the ‘building blocks’ of systems so any dependability assessment that targets them will be of wide use, not only to Web Services but also to Globus Grid services, which utilize the same technology.

Both WSDL and SOAP utilize eXtensible Markup Language (XML) [8] to define and implement Web Service message exchanges. XML is a standard for document markup. It provides a document layout that allows a document to be self-describing and portable,
allowing data transfer between dissimilar systems. Its portability is largely due to it being an ASCII format document, with numeric values encoded as strings. Since it is portable it largely eliminates the need for marshalling and unmarshaling of data, but any reduction in overhead is more that offset by the need to construct XML documents and parse them at the receiving machine, which introduces a greater overhead.

XML is flexible and can be used to represent a large variety of data but most programs constrain this flexibility. A Document Type Definition (DTD) defines which elements are permissible and the circumstances in which these elements can be used under. This allows the basic structure and syntax of an XML document to be defined and validated. An XML schema can be used to describe complex restrictions on a document, such as type information, and complex data types. It is possible to validate an XML document to see if it follows the rules defined in DTDs and schemas. SOAP uses a collection of schemas to define a standard set of types that can be used as well as defining the message structure, etc.

Web Services present a defined interface to utilizing applications and this is constructed by the use of WSDL. This is an XML-based Interface Definition Language (IDL) used to define Web Services interfaces and how to access them [7]. Our research is mainly concerned with RPC message exchanges and WSDL lends itself well to providing explicit information on the structure of message exchanges between Web Services and their clients.

Communication between a utilizing application and a Web Service is usually achieved using SOAP, which is a messaging protocol designed to allow the exchange of messages over a network. It is XML-based to allow the exchange of messages between heterogeneous machines.

Although this work is primarily concerned with the RPC mechanism over SOAP, most of the concepts apply to document oriented patterns of communication using SOAP as well.

3. Grid-FIT

Fault Injection Technology (FIT) is our network level fault injector framework designed to work with middleware systems [5]. FIT contains a Fault Injection Engine (FIE) that is implemented so that different middleware message formats can be handled, including both text and binary. Grid-FIT is a specific tailoring of the FIT framework to work with Globus Toolkit 4. A similar tailoring was used to produce the WS-FIT tool. Grid-FIT is implemented as a plug-in for Eclipse, which is a platform independent framework for developing applications (see Figure 1).

The major innovation of Grid-FIT is the novel fault injection mechanism that allows network level fault injection to be used to simulate Code Insertion fault injection whilst circumventing the need for modifications to the service source code [6]. This is accomplished by intercepting middleware messages within the protocol stack, decoding the middleware message in real-time and injecting appropriate faults. By decoding the middleware message and allowing this level of targeted fault injection, it is possible to perform parameter perturbation similar to that achieved by Code Insertion at the API level, and this can also be used to perturb SOAP element attributes in order to assess middleware protocols.

![Figure 1: The Eclipse Based Grid-FIT Tool](image)

Standard network level fault injection works by performing operations on network packets at the physical network interface. Since the fault injection is done at the network interface, modifications to these packets tend to only be reflected at the middleware level as random corruption of data; even reordering and dropping of packets may only result in corruption of a data stream, since a middleware level message may span more than one physical network packet. Further more, reordered or dropped packets may be subject to error correction such as retransmission so faults injected may not reach the middleware layer. Finally, packets corrupted at this level may be rejected by the network protocol stack, for instance via mechanisms such as checksums. It is thus hard to target a particular element of a middleware message with any great certainty. Therefore network level fault injection has traditionally been used only for assessing network protocol stacks, not service based systems.

The FIT method of network level fault injection takes the basic concept given above but moves the fault
injection point away from the network interface and positions it in the actual middleware transport layer (see Figure 2). Since middleware messages are then intercepted as complete entities, it is possible to corrupt, reorder and drop complete messages, rather than just part of a network packet that may be discarded before it reaches the middleware layer. Messages can thus be modified and then passed on to the rest of the protocol stack. In this way faults can be injected but not filtered out by the protocol stack.

![Figure 2: Grid-FIT Injection Points](image)

Further, if the messages are intercepted before they are signed or encrypted (or after they are decrypted and the signature checked in the case of incoming messages), individual elements can be corrupted within a message without that message being rejected by the middleware as having been tampered with. Since we can assume we are familiar with the rules and metadata used to construct messages for the specific middleware we are using, by combining the corruption of data in a message with these rules and metadata, it is possible to produce meaningful perturbations of such things as RPC input parameters, so we can use our network level fault injection method to simulate API level fault injection.

![Figure 3: System Model](image)

The rules and metadata used to define the interface of a service are contained within the WSDL definition for that service. This information can be interpreted to decompose the service interface into method calls with their associated messages and, within the messages, identify specific parameters. The FIT method decomposes this information into a taxonomy called the System Model (see Figure 3) which provides all the information required to construct fault injection triggers as described above.

By using this taxonomy and creating triggers on specific elements, the FIT method can precisely inject specific faults rather than random faults into middleware messages as in standard network level fault injection techniques. The method will decode the middleware message and inject meaningful faults, such as modifying RPC parameters and results, adjusting element attributes, etc. The method builds on the FIT framework to allow test cases to be written. These test cases can either be written manually, or automatically generated using the FIT Extended Fault Model (EFM), which is an extensible toolkit of Fault Models that can be applied to parameters and messages [5].

Since FIT can perturb individual RPC parameters within middleware messages, it is particularly well suited to assessing systems by substituting invalid values in place of valid ones and thus testing such mechanisms as guard code at calling interfaces. Whilst it can be argued that the value of assessing interactions between calling interfaces is of limited use in normal non-distributed programming, since these APIs will only be called under known conditions [9], this does not necessarily hold for distributed systems since the interfaces are exposed and can be combined with other logic in unforeseen ways. Assessment of validation mechanisms is therefore key to the production of robust services. FIT has been successfully used to not only assess service interfaces but also to assess third party dependability means [10].

Grid-FIT decodes incoming middleware messages via a SAX parser. Whilst this is an overhead it has been found to be acceptable when compared to network transfer times commonly encountered in Grid applications, and the method has been successfully applied to latency injection test cases [11].

4. Case Study

This case study applies the Grid-FIT tool to Globus Toolkit in order to demonstrate how dependable Apache Axis is in terms of integrity. This is a key concern since Apache Axis is the middleware layer that Globus Toolkit is built upon, and hence has a direct bearing on the amount of trust that can be placed on Globus Toolkit. The case study examines the Axis SOAP stack in terms of corruption of SOAP messages and compliance with the W3C specifications.

4.1. Configuration

The case study examines the effect of fault injection on a representative sample of types defined by the xsd schema. To do this a simple service was
written which included a method for each xsd type. Each routine received a specific xsd type and echoed it back unchanged as the return value. A test program was written which called each service method in turn with a valid instance of the type and compared the return result with the original data sent. In each case the value returned should be identical with that sent. The combination of the service and the test program provide a simple test bed to test xsd types.

The case study was preformed using the same version of Apache Axis as that included in Globus Java WS Core 4.0.3, namely Apache Axis 1.2RC2 1242 April 28 2006. Eclipse 3.2 was used to execute the Grid-FIT plug-in and all software was executed under Mac OS X 10.4.8 running on a PPC architecture.

4.2. Baseline Experiment

A baseline experiment was undertaken to determine the normal operating conditions of the test program and service. The results of this experiment are given in Table 1 which shows two criteria for each xsd type assessed: 1) the comparison of the returned value with the sent value; and 2) any exception that was generated as a result of the method call.

In general the middleware behaved as expected, and from Table 1 it can be seen that none of the types tested generated an exception. An unexpected outcome of this experiment was the Date and DateTime returned values did not match the original values sent.

Date and DateTime are implemented using the Java Standard Library Date class. Equality between two Date instances is obtained only if they match to the millisecond. Examination of the SOAP messages exchanged and the W3C specifications show that the ASCII format of Date passed within the SOAP message does not specify Date to the millisecond, so when they are passed into the Java Date class a slight discrepancy is introduced, hence the returned instance will not match the one originally sent.

The xsd types defined in Table 1 can be grouped into three groupings: 1) xsd types mapped to built-in Java types (dark grey in table) which comprise double, int, boolean, byte, float, long and short; 2) xsd types mapped to Java Standard Library classes (light grey in table) which comprise String, Date, DateTime, Decimal, QName and AnySimpleType; and 3) xsd types that require specially written classes within Apache Axis (white in table) which comprise AnyURI, Duration, GDay, GMonth, GMonthDay, GYear, GYearMonth, Language, Name, NCName, NegativeInteger, NMTOKENS, NonNegativeInteger, NonPositiveInteger, NMToken, NormalizedString, NOTATION, PositiveInteger, Time, Token, UnsignedInt, UnsignedByte, UnsignedLong and

<table>
<thead>
<tr>
<th>xsd type</th>
<th>Returned value equals sent value</th>
<th>Exception Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map to built-in Java type</td>
<td>TRUE</td>
<td>none</td>
</tr>
<tr>
<td>Map to Standard Java Library Class Date</td>
<td>TRUE</td>
<td>none</td>
</tr>
<tr>
<td>Map to Standard Java Library Class DateTime</td>
<td>FALSE</td>
<td>none</td>
</tr>
<tr>
<td>Map to Apache Axis Class</td>
<td>TRUE</td>
<td>none</td>
</tr>
</tbody>
</table>

4.3. Protocol Invalidation

This experiment attempts to invalidate the SOAP protocol whilst retaining syntactically correct XML. This technique can be used to determine if the implementation follows the protocol specification.

Thompson et al [12] specify that “An element may be valid without content if it has the attribute xsi:nil with the value true. An element so labeled must be empty, but can carry attributes if permitted by the corresponding complex type”. Since the test program generates SOAP messages that contain data within the elements, if xsi:nil=”true” is added as an attribute to the part it should fail the scheme validation. By this definition, a SOAP message exchanging a non-null parameter should not contain the xsi:nil attribute.

A fault model was constructed to add xsi:nil=”true” to an element. It was then applied to each request message parameter element. This generated SOAP messages containing this attribute (see Figure 4).

![Figure 4: Modified SOAP message](image)

Table 2 shows the results from this experiment. The results show that only certain types generated an exception, and these did not seem to be descriptive of the schema validation. Of the types that returned a value, the value returned was null. We can therefore conclude that setting xsi:nil, rather than causing an XML schema violation, is implemented by the middleware to infer that the element is empty and any
contents should be silently discarded. This causes a null object to be passed to the service and this null object is passed back in the normal way. We have encountered this type of behaviour in previous case studies with previous versions of Apache SOAP [13].

The exceptions generated can be explained by the \textit{xsd} type implementations in Java being classified into two distinct groupings: 1) mapping to built-in types; and 2) implemented through Java classes, either standard Java classes or specifically written.

The groups that do not generate any exceptions are groups of \textit{xsd} types that map to Java classes, and therefore the null parameter can be passed as a valid parameter. This indicates that no schema validation is explicitly performed. The group that generates exceptions (the dark grey shaded group in Table 2) map to built-in types. These types cannot assume a null value in Java, so the implementation is mapping the null value to a generic Java \textit{Object} and attempting to match to \textit{method( Object)} which does not exist in the service; consequently the misleading exception is thrown. It can therefore be concluded that the exception is being thrown as a consequence of executing an unexpected control pathway, rather than a deliberately implemented piece of guard code in the middleware.

**Table 2: Protocol Invalidation Results**

<table>
<thead>
<tr>
<th>\textit{xsd} Type</th>
<th>Returned Value</th>
<th>Exception Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map to built-in Java type</td>
<td>No result returned</td>
<td>No such operation</td>
</tr>
<tr>
<td>Map to Standard Java Library Class</td>
<td>FALSE</td>
<td>None</td>
</tr>
<tr>
<td>Map to Apache Axis Class</td>
<td>FALSE</td>
<td>None</td>
</tr>
</tbody>
</table>

**4.4. Injecting Bad Data**

This experiment examines two \textit{xsd} types when invalid data is injected. It examines whether ASCII characters which invalidate the schema for the \textit{xsd} types are detected by the middleware and rejected.

![Figure 5: Language message after injection](image)

The first type is an \textit{xsd} \textit{language} type. A valid \textit{language} type has strict rules about the data that can be encoded with it. The encoding follows a schema that defines the types and positions of ASCII characters that can be included in this element. Figure 5 shows the message after the fault is injected. The new element contents are invalid since both the characters contained and the placement of the characters are invalid.

The unmarshaling of this invalid data did not generate any exceptions (see Table 3). The only consequence of this injection was that the returned value was not equal to the original value. Since this type is built upon a string type, it is reasonable to assume that the unmarshaling process assumes that the input data is valid and inserts it directly into the class instance without validating it. Conversely a check of the Apache Axis class that implements this type in Java shows that validation is done when the normal constructors are used.

The second \textit{xsd} type assessed was \textit{PositiveInteger}. When bad data was injected into this message exchange a \textit{Number Format Exception} was generated. Whilst this would appear to be a valid exception closer inspection of the code revealed that this exception was generated as part of the standard Java string to number parsing mechanism, not as part of an explicit validation mechanism. This exception can therefore be considered an interaction fault.

**Table 3: Invalid data Results**

<table>
<thead>
<tr>
<th>\textit{xsd} Type</th>
<th>Returned Value</th>
<th>Exception Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>FALSE</td>
<td>None</td>
</tr>
<tr>
<td>PositiveInteger</td>
<td>No result returned</td>
<td>Number Format Exception</td>
</tr>
</tbody>
</table>

**4.5. Invalid by Omission Faults**

This experiment injected syntactically correct data into the elements containing \textit{xsd} types, but in one case the data invalidated the type’s schema by omission (it should have started with a minus sign). This test gave an appropriate exception (see Table 4) but, as above, this was generated by the Java class parsing the value, rather than a direct validation against the schema.

**Table 4: Invalid by Omission Results**

<table>
<thead>
<tr>
<th>\textit{xsd} Type</th>
<th>Returned Value</th>
<th>Exception Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>NonNegativeInteger</td>
<td>FALSE</td>
<td>None</td>
</tr>
<tr>
<td>NonPositiveInteger</td>
<td>No result returned</td>
<td>Number Format Exception</td>
</tr>
</tbody>
</table>

The nonNegativeInteger was also injected as a control and since the value 128 was a correct value for this type the only consequence of this injection was that the returned value did not match the originally sent value.
5. Conclusion and Future Work

This paper has detailed the application of our GridFIT tool to the Apache Axis component of the Globus Toolkit to provide dependability assessment with regards to its integrity.

Our case study uncovered a number of discrepancies between the Apache Axis implementation and the W3C SOAP specification. These fall into two categories: 1) misinterpretation/bad implementation of the SOAP specification; and 2) failures due to lack of validation.

The first category includes the xsi:nil misinterpretation of the specification. Whilst this would appear to be a fairly minor fault, it could conceivably form the basis of a buffer overflow attack or denial of service attack since large quantities of data could be transferred in a message and Axis would not flag this as an error.

The second category of discrepancies allows unanticipated control pathways to be exercised that could lead to interaction faults being generated. A closer examination of this would be required on a system-by-system basis to ensure that this could not be employed for compromising the integrity of a system.

We assume that the potential to exploit these particular discrepancies is fairly low in terms of a Globus Grid, since it will be working with known implementations of SOAP stacks and systems will be secured, but the potential does exist to exploit them by using modified middleware stacks since validation is undertaken by guard code which appears to be contained in the Java class implementations of the xsd types, not as a general message validation mechanism.

Future work will concentrate on enhancing our fault models to more thoroughly examine the Grid middleware, determine if Grid-FIT can be successfully applied to assess other dependability attributes, and apply our technique to more complex Globus systems and scenarios.

6. Acknowledgments

This work was funded as part of the EPSRC funded CoLab project (EPSRC Reference: EP/D077249/1). Thanks are due to Dr Paul Townend for his help and suggestions in the preparation of this paper.

7. References