Application of Fault Injection to Globus Grid Middleware

Nik Looker (a), Jie Xu (a), Tianyu Wo (b), Jinpeng Huai (b)

(a) University of Leeds, Leeds. LS2 9JT, UK
(b) Beihang University, Beijing 100083, PRC

Abstract

Dependability is a key factor in any software system and has been made a core aim of the Globus based China Research Environment over Wide-area Network (CROWN) middleware. Our past research, based around our Fault Injection Technology (FIT) framework, has demonstrated that Network Level Fault Injection can be a valuable tool in assessing the dependability of RPC oriented SOAP based middleware. We present our initial results on applying our Grid-FIT method and tools to Globus middleware and compare our results against those obtained in our previously published work on Web Services. Finally this paper outlines our future plans for applying Grid-FIT to CROWN and thus providing dependability metrics for comparison against other middleware products.

1 Introduction

The Globus Toolkit [1] is an open source software toolkit used for building Grid systems and applications [2]. Since it is the front running technology for Grid computing its dependability is a key issue. A large part of the infrastructure for Globus Toolkit 4 is constructed around Java Web Services utilizing Apache Axis [3] as the transport and Apache Tomcat [4] as the container.

Fault Injection is a well-proven method for assessing the dependability of a system. Although much work has been done in the area of fault injection and distributed systems in general, there appears to have been little research carried out on applying this to Web Service based systems [5, 6]. Network Level Fault Injection has provided promising results in assessing the dependability of SOA [7].

Web Service – Fault Injection Technology (WS-FIT) [7] is a dependability assessment method and tool for assessing Web Services by fault injection. WS-FIT is targeted towards Java Web Services implemented using Apache Axis transport. Given the similarities in underlying middleware technology our aim has been to implement a second tool, Grid-FIT, targeted at Globus Grid Services.

This paper details the fundamental concepts behind our method and outlines the differences between WS-FIT and Grid-FIT. We demonstrates Grid-FIT with a number of proof of concept experiments and compare the results against similar results obtained from WS-FIT. Finally we outline our future plans to apply Grid-FIT to a middleware product implemented over Globus Toolkit to provide data that will allow us to construct a fault model and failure modes to assess Grid middleware and also provide dependability metrics for the middleware products.

2 Service Technology

Globus is built upon Web Service middleware and FIT manipulates messages exchanged between Web Services to assess service dependability.

A Web Service is a software service defined by a number of standards that can be used to provide interoperable data exchange and processing between dissimilar machines and architectures. For the purposes of our research we are concerned with Web Services defined by the W3C that are described by WSDL [8] and implemented using SOAP [9] and the RPC model described in that document.

2.1 WSDL

Web Services Description Language (WSDL) is an XML-based Interface Definition Language (IDL) used to define Web Services and how to access them [8, 10]. Our research is mainly concerned with RPC message exchanges. WSDL lends itself well to providing explicit information on the structure of message exchanges between Web Services and their clients. WSDL documents are made up of a number of XML elements and this gives us a good starting point for producing fault injection triggers.

Table 1 shows an example wsdl:message. A wsdl:message is composed of an element that has a unique name attribute that is used to identify the message and a number of wsdl:part
Each wsdl:part defines a parameter (or return value in the case of a response message). A wsdl:part has an associated name that must be unique within the wsdl:message element and a type that defines the parameter type. There are a number of predefined types and complex types can also be defined using a types element.

```xml
<wsdl:message name="unregisterServiceRequest">
  <wsdl:part name="context" type="xsd:string"/>
  <wsdl:part name="entry" type="impl:ServiceEntry"/>
</wsdl:message>
```

**Table 1: Example WSDL Message**

Once all request and response messages required to implement an RPC style interface have been defined they can be used to define the calling interface for the Web Service. This is done by use of the wsdl:portType element (see Table 2). A wsdl:portType contains a number of wsdl:operation elements with each wsdl:operation element corresponding to a method of the Web Service. Each wsdl:operation is made up of a wsdl:input element that will be the request part of the RPC and a wsdl:output element that will be the response message of the RPC.

```xml
<wsdl:portType name="Quote">
  <wsdl:operation name="getQuote" parameterOrder="symbol">
    <wsdl:input name="getQuoteRequest" message="impl:getQuoteRequest"/>
    <wsdl:output name="getQuoteResponse" message="impl:getQuoteResponse"/>
  </wsdl:operation>
  <wsdl:operation name="unregisterService" parameterOrder="context entry">
    <wsdl:input name="unregisterServiceRequest" message="impl:unregisterServiceRequest"/>
    <wsdl:output name="unregisterServiceResponse" message="impl:unregisterServiceResponse"/>
  </wsdl:operation>
</wsdl:portType>
```

**Table 2: Example WSDL PortType**

The above explanation briefly describes the use of WSDL to define a classic RPC style Web Service. WSDL can also be used describe other styles of Web Service calling interface such as message oriented calling but this is outside the scope of this research.

### 2.2 SOAP

SOAP [9, 10] 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 dissimilar machines.

Our method is primarily concerned with RPC mechanisms over SOAP. This is defined by the W3C in [11] and describes a general purpose RPC mechanism. The message types that are involved in an RPC exchange and the relevant features used by our method are briefly reviewed here.

A SOAP message utilizes the http://schemas.xmlsoap.org/soap/envelope/ schema which defines the namespace soapenv and this namespace is setup in the soapenv:Envelope element. Consequently all elements that unitize this namespace must be enclosed by the soapenv:Envelope element. The soapenv namespace defines a semantic framework for SOAP messages.

The body of the SOAP message is enclosed by the soapenv:Body element. This element acts as a grouping for the body elements for different types of messages. Its primary function is to keep the body elements distinct from other grouping of elements, for instance a header block.

These two elements form the basis of a SOAP message. The soapenv:Body element is then populated with elements that make up the payload of a request, response or fault message.

A typical request message is given in Table 3. The request message name is defined in the wsdl:operation (see Table 2) but by convention the name of the message equates to the service method name but it can be defined as any valid name. In the example the message and method name are getQuote. The message element is therefore ns1:getQuote. The namespace ns1 is defined to be the urn of the service that implements the method. If this is combined with the address of the server hosting the service this allows a specific method of a specific service on a specific server to be identified.

The ns1:getQuote element contains parameter elements that represent the RPC parameters, for instance the getQuote method takes one string parameter called symbol so ns1:getQuote contains one element with an element tag symbol which contains the string data for that parameter. Parameters are defined in WSDL by wsdl:part elements in wsdl:message elements (see Table 1).
A typical response message is given in Table 4. The response message is similar in structure to the request message but by convention the response message name is post-fixed with the word Response although, again, it can be any valid name defined in the wsdl:operation element. In this example the response element name is ns1:getQuoteResponse.

```xml
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:getQuoteResponse
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/>
      xmlns:ns1="http://quote.stock.samples.dasbs.org">
      <getQuoteReturn href="#id0"/>
      <multiRef id="id0" soapenc:root="0"
        soapenv:encodingStyle="http://quote.stock.samples.dasbs.org">
        <date xsi:type="xsd:dateTime">
          2004 10 30T10:54:18.511Z
        </date>
        <quote xsi:type="xsd:double">
          47.5
        </quote>
      </multiRef>
    </ns1:getQuoteResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Table 3: Typical Request Message

A response element contains elements that represent any method return value and any parameters that are marked to be marshalled in-out or out. Method return results follow the naming convention of the method name post fixed by the word Return and are represented in the WSDL by wsdl:part elements. In-out and out parameters follow the same conventions as parameters in a request message.

The example response message in Table 4 also demonstrates the format that objects and arrays are marshalled in a SOAP message. This utilizes the multiRef element. Each object or item in an array is created using a multiRef that has an id. The actual parameter/return value is then set to this reference id and the complex data can be constructed within the multiRef element in the same way that individual parameters are constructed in a message. This technique applies to both request and response messages.

Table 4: Typical Response Message

```xml
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <soapenv:Fault>
      <faultcode>
        soapenv:Server.userException
      </faultcode>
      <faultstring>
        java.rmi.RemoteException: can't get a stock price
      </faultstring>
    </soapenv:Fault>
  </soapenv:Body>
</soapenv:Envelope>
```

Table 5: Typical Fault Message

3 Grid Middleware

Grid middleware allows the construction of Grid services. It typically provides a transport mechanism as well as security facilities that give a consistent set of operations over a heterogeneous platform base.

Globus Toolkit is the front running reference implementation of OGSA. Version 4 of this product is based around a Java core composed
of Apache Tomcat and Apache Axis that makes the Globus environment similar to our previous experimental environments.

The major difference is the way that Globus packages and exchanges data. Whilst Globus utilizes the standard Axis RPC mechanism it packages parameters and return values into a complex data structure. This means that each message exchange has a schema associated with it (See Table 6) rather than each parameter being defined as a set of wsdl:part in the wsdl:message element.

```xml
<xsd:element name="getQuote">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="symbol" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Table 6: Example Globus WSDL

This means that the WSDL for each request and response message has a more nested structure but this has no major effect on the structure of the messages exchanged. This is because when there is only one object passed as a parameter it can be contained in the message body rather than as a sequence of multiRef elements (See Table 7).

```xml
<soapenv:Body>
  <getQuote xmlns="http://...">
    <symbol>foo</symbol>
  </getQuote>
</soapenv:Body>
```

Table 7: Example Globus SOAP Message

Whilst some modification to the code base is necessary to accommodate these differences in WSDL they are fairly minor.

3.1 CROWN

To fully evaluate our system we require a complex application to access. In doing so we would gain useful knowledge on how to construct fault models and where to apply them in a system. With this in mind we have selected a test bed system which will provide us with a real-world test scenario.

CROWN is a Grid test bed to facilitate scientific activities in different disciplines. The CROWN middleware platform is consist of 3 parts. Firstly, many resources (including computers, clusters and some storage devices) should be connected via a nation-wide network infrastructure to build the CROWN architecture. Secondly, a series of grid middleware and auxiliary tools should be provided to meet the common requirements of different scientific activities in all kinds of areas. Finally, a number of typical applications should be put in use over the fabric and middleware to demonstrate the feasibility and robustness of CROWN.

The CROWN NodeServer is the basic environment for Grid services to be deployed and executed in the CROWN system. It has 5 main features that are outlined here.

- **Grid service container:** Based on the Globus Toolkit WS Core 4.0, CROWN NodeServer implements a fundamental Grid service hosting environment that follows the OGSA/WSRF specifications.
- **Remote and hot service deploying:** The CROWN NodeServer allows service providers to deploy and configure Grid services remotely without having to restart the container. This is combined with a security and trustworthy mechanism to provide a way for distributing applications through the CROWN system to execute securely and enhances the scalability of the whole system.
- **Resource monitor:** CROWN NodeServer gathers both static and dynamic information of the underlying resource, and shares them via a Grid service interface. The information can be used to control workflow or monitor a system.
- **Legacy applications integration:** Many applications are not developed in a service oriented way. To add them into Grid environment, CROWN NodeServer can encapsulate these programs as Grid services.
- **Security and trust:** The NodeServer follows the WS-Security specification and provides security functions of SOAP signature and encryption, authentication, authorization and identity mapping between different security infrastructures. By using automated trust negotiation, NodeServer can help to setup trust relationship among strangers.

By using NodeServer, we encapsulated heterogeneous resource into a single homogeneous view. All kinds of resources in CROWN are provided via the services running in NodeServer.

We have selected the NodeServer as our first real-word Grid-FIT test case because it gives us a complex system to test, is written utilizing SOAP and it’s assessment would be a benefit to both parties involved in the collaboration, i.e. The middleware can be made more reliable and Grid-FIT can use the data gathered to construct better fault models.
4 Grid-FIT Method

Grid – Fault Injection Technology (Grid-FIT) [6, 7, 12, 13] is a tool derived from our WS-FIT tool and is a fault injector that allows network level fault injection to be used to assess Grid middleware. Grid-FIT has been implemented specifically to assess Globus systems. The implementation specifically handles the problems associated with modifying messages when signing and encryption are being used [14].

Perturbation [15] attempts to forcefully modify program states without mutating existing code statements. This is often achieved by code insertion. Instrumented code (termed perturbation functions) is added to a system in the form of function calls that modify internal program values. These modified values can either be generated based on the original value, generated randomly or a fixed constant value can be used. This technique is useful in testing such things as fault tolerance mechanisms. Our method extends this method to make meaningful perturbation to a SOAP message, e.g. our method can target a single parameter within an RPC message sequence. Since Grid-FIT can effectively achieve this without the need for modification or access to the Service source code it is less invasive that code insertion [16].

Our method uses an instrumented SOAP API that includes two small pieces of hook code. One hook intercepts outgoing messages, transmits them via a socket to the fault injector engine and receives a modified message from the fault injector. This message is then transmitted normally to the original destination. There is a similar hook for incoming messages, which can be processed in the same way (See Figure 1).

Our triggering mechanism is based around specific messages being received. These messages can further be decomposed to specific parameters within a specific message so a particular fault model can be applied to a specific parameter.

Triggers are constructed based on the WSDL for a Web Service (see Section 2.1). The WSDL definition for a Web Service contains detailed information on the messages exchanged. The GUI can import WSDL and parse the wsd:message, wsd:portType and wsd:operation elements into a taxonomy that can be used to construct triggers. By mapping the wsd:message elements onto soapenv:body elements in a message stream and further mapping wsd:part elements onto RPC parameters contained in a soapenv:body we can construct precise triggers that allow perturbation of individual parameters. This indicates the required part of a SOAP message and a fault can be injected at this point such as perturbing a parameter.

5 Case Study

The purpose of this case study is to demonstrate that Grid-FIT operates in a similar fashion to the WS-FIT method, whilst not causing any unintentional degradation in the operation of the Globus middleware. To achieve this we have adapted a previously experiment based around WS-FIT and Web Services so that the results can be compared [17].

In Looker et al [17] these tests were constructed using the application of the Extended Fault Model (EFM) implemented by FIT but due to time constraints the results given here were obtained by manually adapting the test scripts rather than reapplying the EFM.

5.1 Test Scenario

To provide a test bed to demonstrate Grid-FIT we have constructed a test system that simulates a typical stock market trading system. This system is composed of a number of elements: 1) A service to supply real-time stock quotes; 2) A service to automatically trade shares; 3) A bank service that provides a simple interface to allow deposits, withdrawals and balance requests; 4) A client to interact with the SOA (See Figure 2).

In our original experiment the services were implemented as Axis 1.1 Web Services running under Tomcat 5.0.28. Our new experiment implements these services as Globus 4.0.1 services although they utilize the same algorithms as before.
We have implemented our stock quoting service to use a large repeatable dataset, stored in a backend database to produce a time based real-time stock quote. Since the quote service is based around a database containing the simulated quote values it is possible to replicate a test run exactly by resetting time etc. to a set of starting conditions.

Our trading service implements a simple automatic buying and selling mechanism. An upper and lower limit is set which triggers trading in shares. Shares are sold when the high limit is exceeded and shares are brought when the quoted price is less than the lower limit.

The buying and selling process involves transferring money using the bank service and multiple quotes (one to trigger the transaction and one to calculate the cost). Since these multiple transactions involve processing time and network transfer time this constitutes a race condition as our quoting service produces timed real-time quotes. Any such race condition leaves the potential for the system to lose money since the initial quote price may be different from the final purchase price. This is intentionally to demonstrate that Grid-FIT does not introduce a significant overhead into the system and thus effect its operation.

This paper details three different series of data: 1) A baseline set of data with the system running normally; 2) A simulated faulty/malicious service 3) A simulated heavily loaded server.

Our test system was implemented using Globus WS-Core Version 4.0.1 running on Apple Mac OS X using G4 1.5 GHz processors and 1Gb RAM.

![Figure 2: Instrumented System](image)

### 5.2 Baseline

Our original baseline experiment using Web Services from the normally running system allowed us to verify that the system operates according to its specification. The system provided matches for transactions 99.8% averaged over a number of runs. The average RPC time was measured at 0.05s and appeared to allow the algorithms to function correctly.

The test case was iterated and the transactions from each were compared. Apart from minor timing variations the analysis showed that the test case was repeatable when Web Service technology was used.

We repeated this experiment with our instrumented Globus system (See Table 8). This gave results similar to the Web Service based system with an average match of 99.5% and an average RPC execution time of 0.05s.

### 5.3 Fault/Malicious Service

The second test series simulated a faulty/malicious quote service by applying a random fault model. The random model used injects a normally distributed randomly generated value that replaces the RPC parameter specified. The random sequence was hard coded into the script to allow repeatability of the test. We used the same starting conditions as the first test series and iterated the test series four times.

For the Web Service based system all transactions produced a mismatch as expected, since each quote value would be corrupted. The Globus based system was then used to run the experiment. This produced a similar result with all transactions being mismatch transactions (See Table 9).
Whilst a detailed investigation of this result is service experiment which gave a result of 63%. To cause the quote to fall into the next quote some quote values being delayed long enough approximately 40% of the time. This is due to shares differs from sale/purchase price the quote value that triggers a sale/purchase of shares differs from sale/purchase price approximately 40% of the time. This is due to some quote values being delayed long enough to cause the quote to fall into the next quote period.

This is significantly smaller that our Web Service experiment which gave a result of 63%. Whilst a detailed investigation of this result is required this could be due to variations in the statically encoded distribution and minor differences in timings introduced by Globus. Small timing variations introduced into this algorithm could produce large effects since each variation would alter a share trading event and influence the amount of money available for the next share trade.

We compared the test runs and the results are given in Figure 5. This shows that whilst the individual test runs vary more that the previous two sets of data they follow the same trend and are repeatable. This is comparable with the experimental data gathered from the Web Service based system.

Table 9: Globus Attack Data

<table>
<thead>
<tr>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match %</td>
<td>61.00</td>
<td>39.00</td>
<td>61.00</td>
</tr>
<tr>
<td>Mismatch Match</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Average Time</td>
<td>0.04</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>Std Dev</td>
<td>3.27</td>
<td>2.02</td>
<td>3.27</td>
</tr>
<tr>
<td>Average Match</td>
<td>60.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Std Dev</td>
<td>0.95743</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: Comparison of Attack Test Data Runs

Again by comparing the Web Service based data with the data obtained in this test case we observed no significant differences.

5.4 Latency Injection

The final series of data in this set of tests again injected a fault into the system. This fault was an increased latency induced into the quote service. This latency simulates server loading. To implement this we introduced a delay into the system based on a poisson distribution. The distribution is statically encoded into the test script to allow for repeatability. The test was iterated over four runs.

Table 10: Globus Latency Data

<table>
<thead>
<tr>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match %</td>
<td>60.00</td>
<td>40.00</td>
<td>60.00</td>
</tr>
<tr>
<td>Mismatch Match</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Average Time</td>
<td>0.04</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>Std Dev</td>
<td>3.28</td>
<td>2.95</td>
<td>3.28</td>
</tr>
<tr>
<td>Average Match</td>
<td>61.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Std Dev</td>
<td>0.95743</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: Globus Latency Comparison

6 Conclusion and Future Work

This paper has detailed the application of our FIT based tool, Grid-FIT, to Globus version 4. We have demonstrated that our dependability method previously developed for Web Service based systems can be applied with little modification to Grid based systems.

We have demonstrated that the differences in definition and message format are negligible with regard to our tool. A minimum of work was required to implement our new Grid-FIT tool from the code base designed for WS-FIT.

Our experimental data from this paper demonstrates that a Grid based system can be assessed using the same method as has been applied in previous research and produces similar results with no apparent unintended impact on the Globus middleware except during the latency injection.

We observed some differences when latency injection was performed between the Web Service based test case and the Globus based test case. This test case is the most sensitive of all tests to timing constraints so a possible cause may be that slightly different timing constrains cause by Grid-FIT and Globus Toolkit caused slightly different control pathways to be executed at different times. Even so this test
case followed the same basic trend as the Web Service based tests.

This work provides us with a proof of concept. As a next step we intend to implement an enhanced fault and failure model to facilitate typical Grid system assessment based on the framework provided by FIT.

To accomplish this we will need experimental data on a non-trivial Globus based system.

We intend to utilize the CROWN middleware to provide us with a test bed system. By applying Grid-FIT to the NodeServer element built into CROWN we will gain valuable experience and data on assessing Globus based systems. This collaboration will be of advantage to both sides since dependability is a key aim of the CROWN middleware and by analyzing the application of our Grid-FIT tool and method to CROWN we will gain valuable data on how to enhance our fault models.

7 References


