

# FT-Grid: A Fault-Tolerance System for e-Science

Paul Townend

Paul Groth<sup>1</sup>

Nik Looker<sup>2</sup>

Jie Xu

School of Computing,  
University of Leeds,  
LS2 9JT, UK

<sup>1</sup> School of Electronics and  
Computer Science,  
University of Southampton,  
Southampton, S017 1B, UK

<sup>2</sup> Dept. of Computer Science,  
University of Durham,  
DH1 3LE, UK

{ pt, jxu } @ comp.leeds.ac.uk

pg03r@ecs.soton.ac.uk

n.e.looker @ durham.ac.uk

## Abstract

*The FT-Grid system introduces a multi-version design -based fault tolerance framework that allows faults occurring in service-based systems to be tolerated, thus increasing the dependability of such systems. This paper details the progress that has been made in the development of FT-Grid, including both a GUI client and also a web service interface. We show empirical evidence of the dependability benefits offered by FT-Grid, by performing a dependability analysis using fault injection testing performed with the WS-FIT tool. We then illustrate a potential problem with voting based fault tolerance approaches in the service-oriented paradigm – namely, that individual channels within fault-tolerant systems may invoke common services as part of their workflow, thus increasing the potential for common-mode failure. We propose a solution to this issue by using the technique of provenance to provide FT-Grid with topological awareness. We implement a large test system, and – with the use of the PreServ provenance system developed as part of the PASOA project at the University of Southampton – perform a large number of experiments which show that a topologically-aware FT-Grid system results in a much more dependable system than any other configuration tested, whilst imposing a negligible timing overhead.*

## 1. Introduction

The approach of design diversity - and especially multi-version design (MVD) - lends itself to service-oriented architectures, as the potential availability of multiple functionally-equivalent services should allow a multi-version system to be dynamically created at much lower cost than would traditionally be the case. At the same time, service-orientation promises to reduce the cost of developing and maintaining any in-house services, as well as the cost of integrating multiple services together [1]. We have constructed an MVD-based weighted fault tolerance framework that enables us to take advantage of the opportunities provided by service orientation for cheaper and potentially better performing fault tolerance schemes, and use the technique of provenance in order to provide both a more finely-grained approach to assessing the dependability of a service, and also to allow a user to tune weightings of services that share common elements in their workflow. Our scheme performs voting on the results of functionally-equivalent services in order to mask design value faults, malicious value faults, and late timing faults. This paper summarises our fault tolerance scheme and our use of provenance to provide topological awareness.

## 2. Service-oriented architectures

A Service-Oriented Architecture (SOA) is an architecture that represents software functionality

as discoverable services on a network. It can be defined as “an application architecture within which all functions are defined as independent services with well-defined invocable interfaces, which can be called in defined sequences to form business processes” [2]. The main principles are not new (CORBA and .NET being examples), but SOAs propose a number of advantages; such as locational transparency, loose coupling, and dynamic composition. Unfortunately, with these advantages come potential problems, not least in the area of dependability [3]. Many traditional distributed systems – such as those in the banking domain – perform computations that require very little execution time (typically no more than a few seconds). However, when considering service-based applications, many B2B (business to business) computations can take hours to perform, whilst scientific grid computations often perform tasks that require several or more days of computation. The execution times involved in service-based applications mean that dependability is a key factor in the success of such applications.

Also, the cost and difficulty of containing and recovering from faults in service-based applications may be higher than that for normal applications [4], whilst the heterogeneous nature of services within an SOA means that many service-based applications will be functioning in environments where interaction faults are more likely to occur. The dynamic nature of SOAs also

means that a service-based application must be able to tolerate (and indeed, expect) resource availability being fluid.

In addition to this, a client application may be able to dynamically locate and utilize potentially non-trusted and external services. Although this late binding at run-time is a useful feature, in many cases at least some of the following properties will hold: 1) We may not be able to be certain that a service is trustworthy (i.e. it will not maliciously alter results); 2) We may not be able to be certain of reliability (either of the service's software or hardware); and 3) We may not be able to be certain that performance criteria are met. It is therefore prudent to look at methods for increasing dependability in such systems; a method of particular interest is that of fault tolerance.

### 3. Fault Tolerance

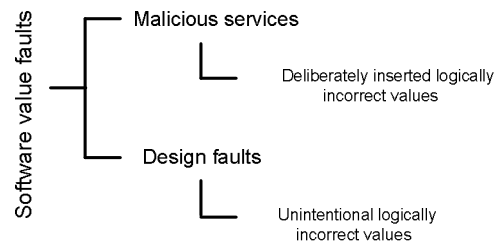
Fault-tolerance has been described by [5] as “...to preserve the delivery of expected services despite the presence of fault-caused errors within the system itself. Errors are detected and corrected, and permanent faults are located and removed while the system continues to deliver acceptable service.” A popular approach when seeking to tolerate faults is that of design diversity, which can be defined the production of two or more systems aimed at delivering the same service through separate designs and realizations. The design-diversity approach that we are particularly interested in is multi-version design (MVD). Traditionally, MVD works by implementing and executing several functionally equivalent systems, comparing their outputs with the consensus output, and forwarding the consensus output as the final system result. This tolerates faults within individual systems, as these will be masked by the voter. Should no consensus be reached, human operators can be alerted to the situation [6].

### 4. FT-Grid

We have constructed an MVD-based fault tolerance framework - called FT-Grid - that will enable us to take advantage of service orientation whilst also facilitating in the creation of dependable service based applications. Implementing an MVD using this framework has a number of advantages, but one of the most important is that of reduced cost of development. In the past, a major obstacle to the use of MVD systems has been the increased expenditure necessary to develop  $n$  functionally equivalent versions of a system (as opposed to one version in a traditional system) and also to maintain each individual version whilst ensuring that its

functionality remains identical to that of the other versions. This problem is much reduced when considering SOAs, as it is possible for a user to be able to dynamically search for, bind, and invoke numerous functionally-equivalent services at run-time. This enables the development of MVD systems to be much quicker and cheaper than has traditionally been the case. This is especially important, as the dependability gain offered by MVD approaches is hard to estimate and quantify, due to the possibility of *common-mode failures* occurring, caused by independent or non-independent faults leading to similar errors between versions of an MVD system. It is therefore harder to justify spending extra resources on the development of such a system, and so reduced development cost makes the approach more justifiable. Because of this reduced cost, it becomes possible for an MVD system to use more versions than is strictly required, in order to reduce latency. For example, should a 3-version voter system be employed, then the performance of the voter system is tied to the performance of the slowest version. However, should a voter system invoke 5 versions and vote on the first three results it receives, then the performance of the slowest two versions is unimportant.

FT-Grid enables voting to be performed on the results of functionally-equivalent services in order to mask faults from the following fault model:



Value faults that result in logically incorrect values can occur as a consequence of either a flaw in the design of a software component or an intentional attack by a malicious service are tolerated by being discarded in the voting process (assuming no common-mode failure occurs). Late timing faults caused by slow service execution due to highly-loaded hosting environments are tolerated as our scheme needs only to wait for a subset of services to finish before voting is performed.

A client application or service instantiates or invokes an FT-Grid service. Although this is conceptually a single service, in reality one may use any of a number of different fault-tolerant

schemes in order to guard against a single point of failure within the system, such as the distributed recovery block approach [7]. The FT-Grid service can be located either on the client side or on a remote machine. A client can use this service to search and retrieve information about available services. The client then selects versions from the services available and passes the relevant input data for the services to the FT-Grid service, as well as specifying what type of voting operation is required, and the number of results to vote on. The FT-Grid service then invokes these services, votes on the results, and returns a single result to the client. Should no definitive result be reached, then FT-Grid can either wait for more results to be received, send out more replicated jobs (preferably to services belonging to organisations different from the services whose results did not reach a majority opinion), or return an error message to the client application. When a majority is reached the result is passed back to the client.

The FT-Grid implementation consists of both a GUI and a web service interface to a library of routines for the facilitation of the MVD fault tolerance approach described above. The FT-Grid GUI allows a user to easily search through any number of UDDI repositories (should more than one UDDI server be specified, then the application will collate and return matching services from all UDDI servers specified), select a number of functionally-equivalent services (this currently a manual task, as automatically detecting functional equivalence is beyond the scope of the scheme at present), choose the parameters to supply to each service, and invoke those services. Services are invoked in parallel and results are received asynchronously. Users can set timeout limits on each service, in order to avoid situations in which results are never returned. In addition to this, users can also specify the number of results to vote on.

There are currently three different kinds of voting available: 1) "consensus" voting, where the majority output is returned; 2) "average" voting, where the average value of all returned numerical results is returned; and 3) "median" voting, where the median value of all the numerical results is returned.

## 5. Dependability Analysis with WS-FIT

In order to test the effectiveness of FT-Grid, we have performed a large amount of fault injection testing, using a tool called WS-FIT (Web Service – Fault Injection Technology) [8]. The WS-FIT tool and method has been developed specifically to perform Network Level Fault Injection and parameter perturbation on SOAP

based SOA. The main use of this technology is state perturbation through the targeted modification of parameters within SOAP messages and the injection of communications faults. This is particularly useful in assessing fault tolerance mechanisms such as FT-Grid. WS-FIT uses a variation of Network Level Fault Injection where faults are injected at the API boundary between the application and the top of the protocol stack rather than at the network interface; this is achieved through the use of an instrumented SOAP API. This eliminates problems associated with altering encrypted/signed messages and allows a fine level of control over message manipulation since SOAP messages can be decoded, thus allowing parameter perturbation.

To provide a test bed to demonstrate FT-Grid we constructed a test system that simulates a typical stock market trading system. This system is composed of a number of Web Services: 1) A service to supply real-time stock quotes; 2) A service to automatically trade shares; 3) A bank service that provides deposits, withdrawals and balance requests. The trading SOA 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 purchased 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 of the transaction. These multiple transactions involve processing and network transfer time and constitutes a race condition since our quoting service produces real-time quotes using a time based algorithm. 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.

We instrumented each server in the system to eliminate any bias caused by slight latency introduced by WS-FIT, although we have demonstrated in previous research that the latency introduced by this instrumentation is negligible when compared to message transfer times involved in a SOAP based SOA [9]. This instrumentation allows us to not only inject faults into the system, but also monitor the RPC exchanges between Web Services. There are three different series of test data: 1) A baseline set of data with the system running normally; 2) A simulated malicious quote service, 3) A simulated heavily loaded server. We used two configurations of the SOA: 1) The SOA running with a single quote service; this will be known as configuration C<sub>1</sub>; 2) The SOA running with 5

	Normal		Malicious service		Heavily loaded	
	C1	C2	C1	C2	C1	C2
Match %	99.50	99.00	0.00	99.71	37.37	76.33
Transaction time (sec)	0.05	0.29	N/A	0.22	3.16	3.28
Transaction time (failure) (sec)	0.06	0.37	0.04	0.20	24.64	11.59
Consensus %	N/A	100.00	N/A	68.38	N/A	50.31

**Figure 1: Results of Dependability Analysis using WS-FIT**

replicated quote services using FT-Grid to provide a fault tolerance mechanism. This will be known as configuration C<sub>2</sub>. C<sub>1</sub> is used to provide a baseline system to compare C<sub>2</sub> against. To demonstrate that our results are repeatable, each test series was repeated five times.

The malicious quote service was created by applying one of WS-FIT's predefined fault model tests to the quote service; the test chosen generated a random value that replaced the RPC parameter returned for a quote; for C<sub>2</sub>, the same test was used and was applied to 2 of the 5 replicated quote services. The heavily loaded server was created by injecting a late timing fault into the quote service; this was done by again using one of the predefined tests in the WS-FIT fault model, to introduce a delay into the system based on a poisson distributed. For C<sub>2</sub>, the fault was injected into 2 out of 5 of the replicated quote services. A summary of the results of these tests are presented in figure 1.

Our analysis of the data demonstrated that for this scenario FT-Grid worked in a non-invasive manner with only a small latency introduced. Whilst this latency is not significant in the context of this demonstration - since timing constraints are in the order of 10s or greater - it could become significant in systems with tighter timing constraints. So although these results are promising, more experimentation is required to determine if this latency is fixed or if it is cumulative, etc.

The operation of the system whilst running under a simulated attack on some of the services showed a marked increase in reliability when using FT-Grid. This is due to the consensus voting mechanism utilized by FT-Grid. This ensures that single corrupted values cannot be passed to the utilizing client. Although there is a drop in the number of readings that are accepted by the client, the overall operation of the system can continue at a degraded level. The degradation was under 10% for this test scenario. Our test system also showed a marked improvement when operating in the FT-Grid configuration when latency was introduced. We observed that whilst the single quote service system was susceptible to latency and an increased number of mismatched readings were observed, FT-Grid saw a reduction in this due again to its consensus voting mechanism. The voting mechanism is designed to vote on the first 3 values returned and discard the other two. Consequently any delayed values are

discarded and the effect of any latency within the quote services is reduced.

Whilst there was a marked increase in the operation of the SOA under test we would have expected an even higher increase. We believe that this is due to an interaction between WS-FIT and Tomcat; since WS-FIT is single threaded with regard to a single Tomcat server it can only process messages from that server sequentially, although it can process data from multiple servers in parallel. Since we are inducing a latency it is possible that in a number of cases WS-FIT is still processing the previous message when the next request by the voter is issued. This would further delay the response from the server to the voter and thus increase the failure rate. We are investigating this effect further but believe it is only present in extreme cases, for instance when a large delay exceeds the frequency of the quotes being issued. Preliminary experimentation with smaller latencies shows a marked improvement which would seem to support this theory.

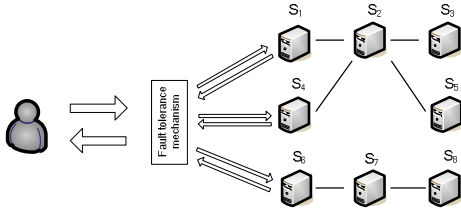
Our fault-injection based dependability analysis demonstrates that for a non-trivial example system, the use of FT-Grid does not adversely effect the operation of the system, provides protection against malicious services, and reduces the susceptibility of the system to timing faults caused by slow running services.

## 6. Provenance to provide topological awareness

A major problem when considering the provision of MVD fault tolerance in service-based applications is that the nature of SOAs is such that a client that invokes a set of services usually does not usually have any knowledge of the underlying implementations of those services. In reality, these services may in turn be composed of other services, some of which are shared between the channel's workflows. An example of this is shown in figure 2.

In this example, a client invokes three seemingly unrelated services – {S<sub>1</sub>, S<sub>4</sub>, S<sub>6</sub>} – with the intention of using their results in a 3-version voter system. S<sub>1</sub> invokes {S<sub>2</sub>, S<sub>3</sub>} as part of its workflow, S<sub>4</sub> invokes {S<sub>2</sub>, S<sub>5</sub>}, and S<sub>6</sub> invokes {S<sub>7</sub>, S<sub>8</sub>}. The invoked services may be running on different machines, with different implementations, and owned by different organisations, or may be services running on a web service farm that is owned by the client but whose job manager load balances the actions

performed by the services, thus resulting in the client not being certain of their exact workflow. This may lead to a situation such as that shown in figure 1, where seemingly disparate services ( $S_1$  and  $S_4$ ) use a common service – in this case  $S_2$  – as part of their workflow. Should an error (caused by, for example, a design fault or a malicious attack) in  $S_2$  lead to a failure in that service, then the effects of this may propagate back to  $S_1$  and  $S_4$ , causing them to both fail in identical ways, and hence cause a common-mode failure.



**Figure 2: Shared service used by two separate workflows**

It can be conjectured that some available services in a given domain may be very popular and hence used by many other services within that domain (and possibly outside of it), thus bringing about uncertainty over the likelihood of common-mode failure occurring when voting on the results of a given set of services. This reduces the confidence that can be placed in the results of design diversity-based fault tolerance schemes.

A promising technique to help us to circumvent this problem is that of *provenance*. The provenance of a piece of data is the documentation of process that led to that data. Provenance can be used for a number of goals, including verifying a process, reproduction of a process, and providing context to a piece of result data. In the context of SOAs, Groth et al. [10] define two types of provenance; interaction provenance and actor provenance. They are defined as follows: for some data, interaction provenance is the documentation of interactions between actors that led to the data. For some data, actor provenance is documentation that can only be provided by a particular actor pertaining to the process that led to the data.

Provenance in relationship to workflow enactment and SOAs is discussed by Szomszor and Moreau in [11]. In a workflow based SOA interaction, provenance provides a record of the invocations of all the services that are used in a given workflow, including the input and output data of the various invoked services. Through an analysis of interaction provenance, patterns in workflow execution can be detected; for example,

one can determine whether a common service was invoked by various other services in a workflow. This data can be used in a fault tolerance algorithm to see if any faults in a workflow stem from the misbehaviour of one service. Using actor provenance, further information about a service, such the script a service was running, the service's configuration information, or the service's QoS metadata, can be obtained. Provenance provides a picture of a system's current and past operational state, which can be used to isolate and detect faults.

## 7. Provenance-aware weighted fault tolerance scheme

Following the completion of our fault injection testing on FT-Grid, we have added additional capabilities to allow the creation of an MVD-based weighted fault tolerance scheme that uses the technique of provenance in order to provide both a more finely-grained approach to assessing the dependability of a service, and also to allow a user to tune weightings of services that share common elements in their workflow.

There exists a plethora of existing schemes to calculate weightings for channels within an MVD system, and such weightings can either be pre-determined or adjusted dynamically. A promising method in the future will be to query a service's QoS metadata in order to generate weightings based on the confidence of that service returning a correct result. However, as no current standards exist on how to represent this meta-data, for this experiment, we generate weightings dynamically, based on a scheme detailed in [12], where historic data of how often a channel's output agrees with the consensus is used to generate history records which are then used in a weighting scheme. However this solution by itself does not deal with problems caused by shared services in the workflow of the MVD channels. In order to resolve this problem, we use a provenance scheme called PReServ [13], which is assumed to be installed on every hosting environment in our system.

*Provenance Recording for Services (PReServ)* is a Java-based Web Services implementation of the *Provenance Recording Protocol* [10]. This defines an implementation independent means for recording interaction and actor provenance in SOAs. Using PReServ, a developer can create a provenance aware SOA by using the following three components provided by the package:

1. A provenance store that stores, and allows for queries of, provenance.
2. A client side library for communicating with the provenance store.

3. A handler for the Apache Axis Web Service container that automatically records interaction provenance for Axis based services and clients by recording incoming and outgoing SOAP messages in a specified provenance store.

By querying the provenance store created by PReServ, it is possible for us to reconstruct the workflow of each channel in our MVD system, such that a service  $i$  invokes  $k$  unique services in its workflow, where  $k \in \{1, 2, \dots, N\}$ . This in turn allows us to adapt the weighting algorithm to take into account this extra information. We do this by keeping a counter  $C_k$  which stores the number of times a service  $k$  is invoked by MVD channel workflows in our system. Should the service  $i$  produce a result that agrees with the consensus result, then every  $S_k$  in that service's workflow is increased by one, else  $S_k$  is set to 0. We can then calculate the weightings of each service  $k$  with

$$P_k = \frac{1}{C_k} \cdot S_k.$$

When a sufficient number of

results have been received to be voted upon, the provenance records of each service are analysed to determine the *degree* of each service  $k$  within the workflows of the services that have returned results; for example, should a service  $k_1$  have a degree of 1, then only one MVD channel has invoked that service. Should  $k_1$  have a degree of 2, then two MVD channels have invoked it during the course of their workflow, etc. We can then further bias the weightings of  $S_k$  based on user-defined settings; for example, should a user specify a bias of 0.95 for a service with a degree of 2, then the final weighting of a service where  $S_k$  has a degree of 2, would be  $W_k = S_k \cdot 0.95$ . Should any service within a given channel fall below a user-defined minimum weighting, then that channel is discarded from the voting process.

## 8. Experiments Performed

In order to test our initial algorithm, we have developed a total of 12 web services spread across 5 machines, each using Apache Tomcat/Axis as a hosting environment, each with provenance functionality, and each registered with a UDDI server. We develop five "Import Duty" services -  $ID_i$ ,  $i = 1..5$ ; - that we use to form a MVD system. An import duty service is designed to calculate the amount of import duty to be paid for a given amount; input parameters are the originating country currency, the destination country currency, and the value of the import. Each import duty service dynamically searches at runtime for both an exchange rate service (in order to convert the import's value to the destination country's currency) and a tax

lookup service (in order to see how much tax is to be paid on this value), and once it has a list of such services, invokes one of each randomly. We provide 4 exchange rate services -  $ER_i$ ,  $i = 1..4$ ; - and 3 tax lookup services,  $TL_i$ ,  $i = 1..3$ ; This system results in a situation where - without provenance data - a client has no knowledge of which exchange rate and tax lookup services are invoked by a given Import Duty service. We simulate a design defect and/or malicious attack by perturbing code in two of the exchange rate services -  $ER_3$  and  $ER_4$  - so that they have a probability of failure (in this case, returning an incorrect value) of 0.33 and 0.5 respectively. We also develop a "perfect", fault-free local version of the services, as a baseline to compare results against. We then perform three experiments on the system, all of which use different aspects of functionality from our FT-Grid system; in all cases, results are then compared against the local fault-free import duty service.

In *experiment 1*, we execute a single version client-side application that invokes a random import duty service, passing it a randomly-generated set of parameters. In *experiment 2*, we execute a client-side MVD application with no provenance capability; this application invokes all 5 import duty services, and waits for the first three results to be returned. The application discards the results of any import duty service whose weighting falls below a user-defined value, and performs consensus voting on the remaining results. Should no consensus be reached, or the number of channels to vote on be less than three, then the client waits for an additional MVD channel to return results, checks the channel's weighting to see whether it should be discarded, and then votes accordingly. This continues until either consensus is reached, or all 5 channels have been invoked. In *experiment 3*, we execute an MVD client-side application with provenance capability. This client invokes all 5 import duty services as in experiment 2. It analyses the provenance records of these channels, and discards the results of any channel that includes a service that falls below a minimum, user-defined weighting. Should no consensus be reached, or the number of channels to vote on be less than three, then the MVD application waits for an additional channel to return results, checks to see if this channel should be discarded, and then votes accordingly. This continues until either consensus is reached, or all 5 channels have been invoked.

## 9. Experimental results

Each experiment iterates 1000 times in order to generate a substantial amount of empirical data.

To demonstrate that our results are repeatable, each experiment is repeated three times. Our test system is implemented using Apache Tomcat 5.0.28 with Web Services implemented using Apache Axis 1.1, and is hosted across 5 dual 3Ghz Xeon processor machines that run Fedora Core Linux 2; our UDDI server is a part of the IBM Websphere package.

### 9.1 Generation of weightings

Using the history-based weighting scheme, we first need some reliable figures to base our weightings on, as weightings maybe be somewhat volatile during initial iterations, and hence unreliable. We therefore initially ran a client application similar to that of our provenance-aware MVD scheme, but which simply generated history weightings based on the consensus results of 1000 invocations of all five import duty services; this application performed no logging or verification of results themselves. From this, we gained the statistics shown in table 1, which we used as starting weights for experiments 2 and 3. As can be seen, the Import Duty and Tax Lookup services have very similar weightings; this is because no faults were injected into these services, but as their weighting depends on results generated from invoking random exchange rate services, their reliability drops below 1, and as the number of iterations progresses, their weightings eventually converge.

ID <sub>1</sub>	0.817
ID <sub>2</sub>	0.872
ID <sub>3</sub>	0.803
ID <sub>4</sub>	0.816
ID <sub>5</sub>	0.838
ER <sub>1</sub>	0.964
ER <sub>2</sub>	0.954
ER <sub>3</sub>	0.713
ER <sub>4</sub>	0.687
TL <sub>1</sub>	0.822
TL <sub>2</sub>	0.825
TL <sub>3</sub>	0.820

**Table 1: Summary of history record-based weightings after 1000 iterations**

Conversely, the weightings of  $ER_3$  and  $ER_4$  show significant deviations from the weightings of  $ER_1$  and  $ER_2$ . This is due to the faults that we injected into  $ER_3$  and  $ER_4$ . It must be remembered that the weights for the  $ER_n$  and  $TL_n$  services can only be acted on by the provenance-aware MVD scheme in experiment 3. Based on the results of these figures, we choose to set the minimum acceptable weighting for our FT-Grid system in experiment 2 and 3 as 0.75. Furthermore, for experiment 3, we define weighting bias of {1.0,

0.98, 0.96, 0.94, 0.92} for services with degrees of {1,2,3,4,5} respectively.

### 9.2 Overall results

Table 2 details the overall findings from this series of experiments. It can be seen that the single-version scheme on average produced a correct result 836 times in every 1000 iterations (83.6% of the time), whilst the traditional MVD scheme produced a correct result 923 times in every 1000 iterations (92.3%), with an average of 66 common-mode failures in every 1000 iterations. The provenance aware MVD scheme produces a correct result on average 994 times out of 1000 iterations (99.4%), and produced no common-mode failures.

	Correct result	No result	CMF
Experiment 1 Run 1	828	172	-
Experiment 1 Run 2	858	142	-
Experiment 1 Run 3	822	178	-
<b>Average</b>	<b>836</b>	<b>164</b>	<b>-</b>
Experiment 2 Run 1	928	9	63
Experiment 2 Run 2	921	14	65
Experiment 2 Run 3	921	7	72
<b>Average</b>	<b>923.33</b>	<b>10</b>	<b>66.66</b>
Experiment 3 Run 1	996	4	0
Experiment 3 Run 2	990	10	0
Experiment 3 Run 3	996	4	0
<b>Average</b>	<b>994</b>	<b>6</b>	<b>0</b>

**Table 2: Summary of data from all three schemes**

### 10. Conclusions and future work

There is increasingly need for solutions that facilitate the provision of dependability in service-oriented architectures. An obvious approach to this is to extend the concept of design-diversity-based fault tolerance schemes, such as multi-version design, to the service-oriented paradigm, as this approach can leverage the benefits of SOAs in order to produce cheaper MVD systems than has traditionally been the case. A major problem however, is that without knowledge of the workflow of the services that form channels within the MVD system, the potential arises for multiple channels to depend on the same service, which may lead to increased incidence of common mode failure.

This paper has proposed a valuable and novel solution which employs – for the first time - the technique of *provenance* to analyse a service’s workflow. We detail an initial scheme that uses provenance to calculate weightings of channels within an MVD system based on their workflow, and we implement a system to demonstrate the

effectiveness of the scheme. In order to test our approach, we develop three different client applications that invoke a suite of services created as a testbed. Our first client application is a simple single-version system that invokes an import duty service and compares the result it receives against a known correct result; we show this system to fail on approximately 16.4% of test iterations. Our second client application performs traditional MVD fault tolerance on the system, invoking multiple import duty services and voting on their results; the result of this voter is then compared against a known correct result. Our results show that dependability is much improved over the single-version system, with an average failure rate of approximately 7.6%; however, this includes an average of over 60 common-mode failures occurring during every 1000 test iterations. Our third client application performs our novel provenance-aware MVD scheme, which can analyse the provenance records of services in order to more accurately ascertain weightings; in addition, services that are shared between multiple MVD channel workflows can have their weightings further biased against. Our test results show that this scheme results in a more dependable system than the traditional approach, with a failure rate of 0.6%, and with no common-mode failures occurring, and we observe that this new approach has a negligible performance overhead over that of the traditional MVD system.

The empirical data we have generated is a valuable first step in evaluating the effectiveness of this approach, and shows that this approach has promise. Future work will include investigation into obtaining QoS indicators from the metadata of each service in an MVD channel's workflow – possibly facilitated through actor provenance – and applying these to the weighting algorithm. We also intend to investigate the relationship between shared components and common-mode failure in more detail, in order to more finely tune our voting scheme.

## 11. Acknowledgements

This research is funded in part by the e-Demand project [14] (EPSRC/DTI Grant THBB/C008/00112C) and the PASOA project (EPSRC Grant GR/S67623/01).

## 12. References

[1] Z. Lin, H. Zhao, and S. Ramanathan, "Pricing Web Services for Optimizing Resource Allocation - An Implementation Scheme," presented at 2nd Workshop on e-Business, Seattle, December 2003.

[2] K. Channabasavaiah, K. Holley, and E. M. Tuggle, "Migrating to a service-oriented architecture, Part 1," IBM Whitepaper, <http://www-106.ibm.com/developerworks/webservices/library/ws-migratesoa/>, 2003.

[3] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," IEEE Transactions on Dependable and Secure Computing, vol. 1, pp. 11-33, 2004.

[4] C. Lee, "Grid RPC, Events and Messaging," GGF APM Research Group Whitepaper, [http://www.eece.unm.edu/~apm/WhitePapers/APM\\_Grid\\_RPC\\_0901.pdf](http://www.eece.unm.edu/~apm/WhitePapers/APM_Grid_RPC_0901.pdf), September 2001.

[5] A. Avizienis, "The N-version Approach to Fault-Tolerant Software," IEEE Transactions on Software Engineering, vol. 11, pp. 1491-1501, 1985.

[6] P. Townend, J. Xu and M. Munro, "Building Dependable Software for Critical Applications: Multi-version Software versus One Good Version", in Object-Oriented Real-Time Dependable Systems (ed. R. Baldoni), IEEE Computer Society Press, pp.103-110, 2002.

[7] K. H. Kim, "The Distributed Recovery Block Scheme," in Software Fault Tolerance, M. R. Lyu, Ed. Chichester: John Wiley & Sons, pp. 189-210, 1995.

[8] N. Looker, M. Munro, and J. Xu, "Simulating Errors in Web Services," International Journal of Simulation Systems, Science & Technology, vol. 5, 2004.

[9] N. Looker, M. Munro, and J. Xu, "Assessing Web Service Quality of Service with Fault Injection," presented at Workshop on Quality of Service for Application Servers, SRDS, Brazil, 2004.

[10] P. Groth, M. Luck, and L. Moreau, "A protocol for recording provenance in service-oriented grids", Proceedings of the 8th International Conference on Principles of Distributed Systems (OPODIS'04), Grenoble, France, December 2004.

[11] M. Szomszor and L. Moreau, "Recording and reasoning over data provenance in web and grid services", International Conference on Ontologies, Databases and Applications of Semantics, volume 2888 of Lecture Notes in Computer Science, pp. 603-620, Catania, Italy, November 2003.

[12] G. Latif-Shabgahi, J.M. Bass, S. Bennett, "History-based weighted average voter: a novel software voting algorithm for fault-tolerant computer systems", 9th IEEE Euromicro Workshop on Parallel and Distributed Processing, Mantova, Italy, pp. 402-409, 2001.

[13] Provenance Recording for Services (PreServ), <http://www.pasoa.org>

[14] The e-Demand Project, <http://www.comp.leeds.ac.uk/edemand/>