

# A Provenance-Aware Weighted Fault Tolerance Scheme for Service-Based Applications

Paul Townend

Paul Groth\*

Jie Xu

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

*\*School of Electronics and Computer Science,  
University of Southampton,  
Southampton, SO17 1BJ, UK*

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

pg03r@ecs.soton.ac.uk

## Abstract

*Service-orientation has been proposed as a way of facilitating the development and integration of increasingly complex and heterogeneous system components. However, there are many new challenges to the dependability community in this new paradigm, such as how 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 scheme that - for the first time - links the technique of provenance with that of multi-version fault tolerance. We implement a large test system and perform experiments with a single-version system, a traditional MVD system, and a provenance-aware MVD system, and compare their results. We show that for this experiment, our provenance-aware scheme results in a much more dependable system than either of the other systems tested, whilst imposing a negligible timing overhead.*

## 1. Introduction

Computing systems in a wide-range of domains are becoming increasingly complex, both in terms of size and also heterogeneity; system components may be implemented using diverse programming languages, on diverse platforms, and may span organisational boundaries. Service-oriented architectures (SOAs), of which Web Services and Grid Computing are implementations, seek to facilitate the development and integration of complex systems by representing software functionality as discoverable services on a network. Given the widespread usage of these technologies, methods for allowing increased system dependability in these environments are greatly desired.

A traditional way to increase the dependability of distributed systems is through the use of *fault tolerant* techniques [1]. The approach of design

diversity - and especially multi-version design (MVD) - lends itself to SOAs, 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 [2].

A potential problem of this approach, however, is that in the traditional SOA model, the implementational details of a service are hidden from a client; the only information available to a client is the service's interface and - possibly - some Quality of Service (QoS) metadata. This may be an issue when developing an MVD system using functionally-equivalent services, as although these services may initially seem disparate (for example, they may be developed or hosted by different organisations), they may - during the course of their execution - invoke one or more identical, "shared" services. Should one of these shared services fail, then the failure may propagate back to the calling services, and result in a common-mode failure (CMF). A CMF occurs when independent or non-independent faults lead to similar errors between versions of an MVD system. Such failures are a "worst case" scenario in a fault-tolerant system, as such failures may be passed through the system undetected; it is often safer to return no result, and alert an operator and/or place a system in a safe state, than it is to allow an undetected error occur.

A solution is offered in the form of *provenance*, which is the documentation of the process that leads to a result. By using a provenance system to record the flow of data from a service, we can identify any shared services, and can use this information when determining weightings for the results returned by each channel. We can then use these weightings to either lower the confidence placed in results shared between  $n$  services, or else discard ( $n-1$ ) of the services from the voting scheme.

This paper describes our novel implementation of a provenance-aware weighted fault tolerance scheme for service-oriented architectures, and details initial results derived from applying this scheme to an example suite of services. To the best of our knowledge, this implementation is the first time that provenance data has been applied to a fault-tolerant approach.

## 2. Dependability in 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*” [3]. However, this paradigm introduces potential problems with regard to the area of *dependability* [4].

When considering service-based applications, many *B2B* (business to business) computations can take hours to perform, whilst scientific grid applications 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.

The cost and difficulty of containing and recovering from faults in service-based applications may be higher than that for normal applications [5] whilst the heterogeneous nature of services within an SOA means that many service-based applications will be functioning in environments where interaction faults (operational faults caused by a system interacting with another system within the use environment) 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 in Service-Oriented Architectures

The function of fault-tolerance has been described by [6] 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 as 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 approach tolerates faults within individual systems, as these will be masked by the voter. Should no consensus be reached, human operators can be alerted [7].

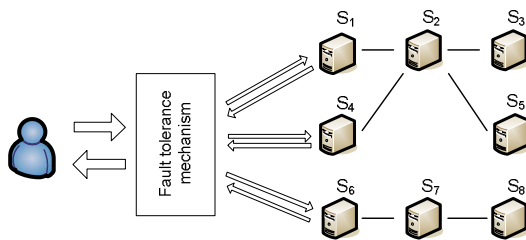
Implementing an MVD system in a service-oriented architecture has a number of advantages but one of the most important is that of reduced cost of development, as it is possible for a user to be able to dynamically search for, bind, and invoke numerous functionally-equivalent services at run-time. This 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 is hard to estimate and quantify due to the possibility of common-mode failure (caused by independent or non-independent faults leading to similar errors between versions of an MVD system) occurring. 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 voter system invoke 5 versions and vote on the first three results it receives, then the performance of the slowest two versions is unimportant.

Unfortunately, the nature of SOAs is such that a client that invokes a set of services usually does not 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

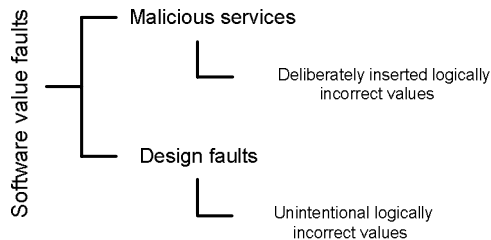
workflows. In the example shown in figure 1, a client invokes three seemingly unrelated services – {S<sub>1</sub>, S<sub>4</sub>, S<sub>6</sub>} – with the intention of using their results in a 3-version voter system. S<sub>1</sub> invokes {S<sub>2</sub>, S<sub>3</sub>} as part of its workflow, S<sub>4</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 where seemingly disparate services (S<sub>1</sub> and S<sub>4</sub>) use a common service – in this case S<sub>2</sub> – as part of their workflow. Should an error (caused by, for example, a design fault or a malicious attack) in S<sub>2</sub> lead to a failure in that service, then the effects of this may propagate back to S<sub>1</sub> and S<sub>4</sub>, causing them to both fail in identical ways, and hence cause a common-mode failure (CMF).



**Figure 1. Shared service used by two separate workflows**

This reduces the confidence that can be placed in the results of design diversity-based fault tolerance schemes, and as far as the authors are aware, until now no work has been performed to assess this problem. A promising technique to help us to circumvent this problem is that of *provenance*, and is detailed in the following section.



**Figure 2. Fault Model**

## 4. 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. [8] 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 [9]. 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. One of our contributions is the use of provenance to assist a fault tolerance algorithm in this manner.

## 5. Provenance-aware Weighted MVD Fault Tolerance Scheme

We have constructed an MVD-based weighted fault tolerance scheme that enables us to take advantage of the opportunities provided by service orientation for cheaper and potentially better performing fault tolerance schemes, and 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. Our scheme performs voting on the results of functionally-equivalent services in order to mask faults from the fault model illustrated in figure 2.

A fault tolerance service – which we call *FT-Grid* – is invoked by a client application or service. Although FT-Grid 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 [10]. The FT-Grid service can be located either on the client side or on a remote machine, and 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 invokes these services, and receives results which are weighted based on the confidence FT-Grid places in each service. Services whose weighting is lower than a user-defined level are eliminated from the voting procedure, and results from redundant versions (should they exist) can be used instead, should their weighting meet the specified criteria. Once voting has been completed, FT-Grid 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 and vote once again, 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 (an action which is preferable to returning an undetected erroneous result, as it allows a system to be placed in a safe state, or an operator alerted, etc.)

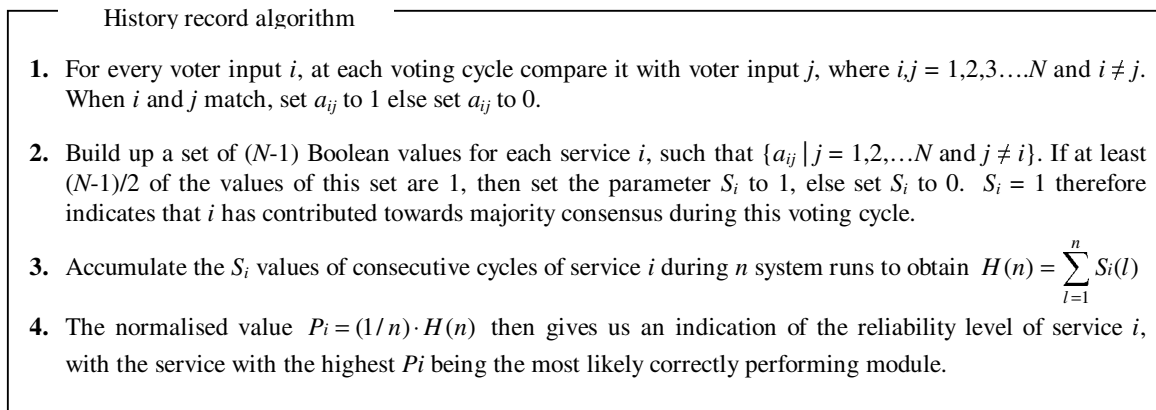
There exists a plethora of existing schemes to calculate weightings for channels within an MVD system, and such weightings can either be pre-determined or adjusted dynamically. A promising method in the future will be to query a service's QoS metadata – possibly stored in the UDDI directory

where the service is located from – 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 [11], 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. This algorithm is defined in figure 3. 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 [12], 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* [8]. 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.

The provenance store component defaults to storing provenance directly on the file system; however, any backend storage system can be plugged into the component. By querying the provenance store created by PReServ, it is possible



**Figure 3. Algorithm to produce history records**

for us to reconstruct the workflow of each channel in our MVD system, such that a service  $i$  invokes  $k$  services in its workflow, where  $k \in \{1, 2, \dots, N\}$ . This allows us to adapt the weighting algorithm above 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 the 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  $P_k = (1/C_k) \cdot S_k$ . 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  $S_i$  has a degree of 2, would be  $W_i = S_i \cdot 0.95$ . Should any service within a given channel fall below a user-defined minimum weighting, then that channel is discarded from the voting process.

## 6. 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 is shown in figure 4.

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.

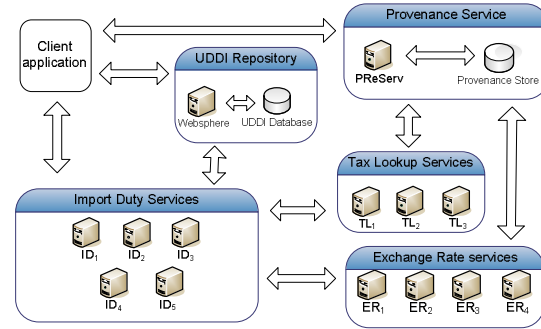


Figure 4. The experimental system

We perform three experiments on the system, all of which use different aspects of functionality from our FT-Grid system; 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. We then compare the result it receives against the fault-free local import duty service, and logs whether or not a correct answer has been returned. In addition to this, the amount of time taken to process the request is also logged.

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. After voting, the weightings of each import duty service are updated dynamically. Results from the voter are then compared against the local fault-free import duty service.

In experiment 3, we execute an MVD client-side application with provenance capability. This client invokes all 5 import duty services, and waits for the first three results to be returned. 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. After voting, the weightings of each import duty service are updated dynamically, as well as the weightings for each exchange rate and tax lookup service. Results from the voter are then compared against the local fault-free import duty service.

## 7. 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.

### 7.1 Generation of weightings

Using the history-based weighting scheme described earlier, we first need some reliable figures to base our weightings on, as weightings may be somewhat volatile during initial iterations. 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 will eventually converge.

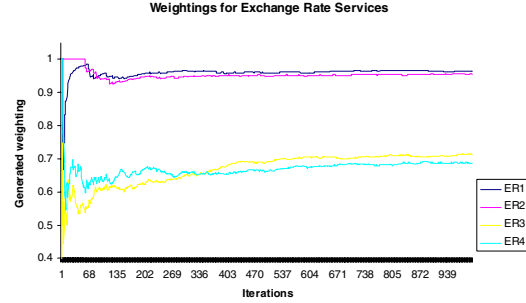


Figure 5. Weightings for exchange rate services over time

Conversely, the weightings of  $ER_3$  and  $ER_4$ , displayed in figure 5, 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.

### 7.2 Experiment 1 – Single version system with no provenance capability

The results obtained by performing the first experiment in our study show – as expected – that the injection of value faults into two of the exchange rate services result in regular failures in the import duty service. Our results are summarised in table 2. As can be seen, over the course of three runs (each of which performs 1000 tests on a random import duty service), an average of 164 incorrect results per run was established; in other words, 16.4% of all invocations returned an undetected, incorrect result.

This is roughly in line with our expectations, as the failure rates of the two faulty exchange rate services lead to the probability of an import duty receiving an incorrect result being:

$$\left(\frac{1}{3} \cdot \frac{1}{4}\right) + \left(\frac{1}{2} \cdot \frac{1}{4}\right) = \frac{5}{24} = 20.83\%$$

Table 1. Summary of history record-based weightings after initial 1000 iterations

ID1	ID2	ID3	ID4	ID5	ER1	ER2	ER3	ER4	TL1	TL2	TL3
0.817	0.872	0.803	0.816	0.838	0.964	0.954	0.713	0.687	0.822	0.825	0.820

We expect over a larger period of time, the failure rate for this experiment would converge on this value. The average time for an import duty to be invoked, and a result received, was 3895 milliseconds; this includes the time taken for an import duty service to query the UDDI repository. This time was quite predictable, with an average standard deviation of 279.72ms.

**Table 2. Summary of results obtained from experiment 1**

	Correct results	Incorrect results	Average time	Standard deviation
Run 1	828	172	3498.809	251.775
Run 2	858	142	4012.901	348.9758
Run 3	822	178	4174.728	238.4373
<b>Average</b>	<b>836</b>	<b>164</b>	<b>3895.479</b>	<b>279.7293</b>

### 7.3 Experiment 2 – MVD system with no provenance capability

The results obtained by performing the second experiment in our study are of interest, as they reveal that even when we enable multi-version fault tolerance in our experimental environment, common-mode failures are frequent. Table 3 shows data from each run of this experiment, and quantifies the outcome of each run of the experiment. For example, in run 1, there were 9 occasions when there were no channels whose consensus agreed with the correct result, whilst there were 604 occasions when the consensus of 3 of the MVD channels agreed with the correct result. Of particular note is the number of common mode failures; for example, there were 63 common-mode failures in the first run - 56 of which came about as a consensus decision by two channels, and 6 of which occurred when all three channels in the

voting mechanism agreed on the incorrect result. This is due to the fact that due to no provenance data being available, each channel had an approximately equal weighting, and hence no unreliable channels were discarded from voting. The average time for an import duty to be invoked and a result received was 4842 milliseconds; this is approximately 1 second longer than when no fault tolerance applied. Like the first experiment, the time taken was quite predictable, with an average standard deviation of 297.53 milliseconds.

### 7.4. Experiment 3 – MVD system with provenance capability

The results of applying provenance information to our fault tolerance scheme show a marked improvement in the detection of common-mode failure. Indeed, in all 3000 iterations across three test runs, not a single common-mode failure occurred, although it must be remembered that in a real-life system, the reliability of services may be much higher than that of the faulty services in this experiment, and so it may be harder to set appropriate discard weightings - and hence some common-mode failures may still occur, although hopefully a reduced amount. The results of this experiment are shown in table 4. As can be seen, the scheme tolerated the vast majority of value faults generated by the faulty exchange rate services; for example, in run 1, of the 1000 tests performed, only 4 resulted in no channels being available for voting (and hence no correct result being provided). The number of channels discarded from the voting process is sometimes quite high (with run 1, for example, discarding 3 of the 5 channels 362 times during the 1000 iterations), but this can be attributed to the low reliability given to the two perturbed exchange rate services. In a more realistic

**Table 3. Summary of results obtained from experiment 2**

<b>Run 1</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
Number of channels agreeing with correct result	9	0	324	604	0	0
Number of non-discarded channels agreeing with correct result	9	0	324	604	0	0
Number of channels discarded	0	0	0	0	0	0
Number of channels used in voting	0	0	0	979	0	21
<i>Number of common-mode failures</i>	-	0	56	7	0	0
<b>Run 2</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
Number of channels agreeing with correct result	14	0	292	629	0	0
Number of non-discarded channels agreeing with correct result	14	0	292	629	0	0
Number of channels discarded	0	0	0	0	0	0
Number of channels used in voting	0	0	0	965	0	35
<i>Number of common-mode failures</i>	-	0	59	6	0	0
<b>Run 3</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
Number of channels agreeing with correct result	7	0	311	610	0	0
Number of non-discarded channels agreeing with correct result	7	0	311	610	0	0
Number of channels discarded	0	0	0	0	0	0
Number of channels used in voting	0	0	0	972	0	28
<i>Number of common-mode failures</i>	-	0	64	8	0	0

**Table 4. Summary of results obtained from experiment 3**

<b>Run 1</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
Number of channels agreeing with correct result	4	4	57	302	325	308
Number of non-discarded channels agreeing with correct result	4	141	362	394	0	0
Number of channels discarded	88	194	211	362	141	4
Number of channels used in voting	0	0	0	88	194	718
Number of common-mode failures	-	0	0	0	0	0
<b>Run 2</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
Number of channels agreeing with correct result	10	12	34	317	324	303
Number of non-discarded channels agreeing with correct result	10	143	327	520	0	0
Number of channels discarded	111	208	201	327	143	10
Number of channels used in voting	0	0	0	111	208	681
Number of common-mode failures	-	0	0	0	0	0
<b>Run 3</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
Number of channels agreeing with correct result	4	13	59	299	317	308
Number of non-discarded channels agreeing with correct result	4	151	336	509	0	0
Number of channels discarded	99	183	227	336	151	4
Number of channels used in voting	0	0	0	99	183	718
Number of common-mode failures	-	0	0	0	0	0

system, the number of channels discarded is likely to be much lower. Another surprising discovery was that this scheme has had no noticeable performance impact over that of the more traditional MVD scheme tested in experiment 2.

The timing data for this experiment reveals that the average time taken to invoke and receive a result using this service was approximately 4751ms – almost 100ms faster than that of the traditional MVD scheme, which had an average time of 4842.639ms, although run 2 of experiment 2 resulted in an average time that was slightly faster than either run 2 or run 3 of the provenance scheme. When taking into account delays introduced by network overhead, the time overhead of the traditional MVD scheme and the provenance aware MVD scheme are approximately the same.

### 7.5. Overall results

Table 5 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

**Table 5. Summary of data from all three experiments**

	<b>Correct result</b>	<b>No result</b>	<b>CMF</b>
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>

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 produced a correct result on average 994 times out of 1000 iterations (99.4%), and produced no common-mode failures.

## 8. Conclusion and further 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 contribution of this paper is to not only detail the potential for provenance data to be used during the voting process of an MVD scheme, but to also implement an initial proof-of-concept for the approach. 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.

## 9. Acknowledgements

This research is funded in part by the e-Demand project (EPSRC/DTI Grant THBB/C008/00112C) [13] and the PASOA project (EPSRC Grant GR/S67623/01).

## 10. References

[1] T. Anderson and P. Lee, *Fault Tolerance: Principles and Practice*. New York: Springer-Verlag, 1990.

[2] 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.

[3] 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.

[4] 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.

[5] 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

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

[7] P. Townend, J. Xu, and M. Munro, "Building Dependable Software for Critical Applications: Multi-version Software versus One Good Version," *IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, Rome, Italy, pp. 103-110, 2001

[8] 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

[9] 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

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

[11] 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

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

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