Increasing Web Service Dependability Through Consensus Voting

Nik Looker
Department of Computer Science, University of Durham, UK
n.e.looker@durham.ac.uk

Malcolm Munro
Department of Computer Science, University of Durham, UK
malcolm.munro@durham.ac.uk

Jie Xu,
School of Computing, University of Leeds, UK
jxu@comp.leeds.ac.uk

Abstract

This paper demonstrates our Web Service based N-Version model, WS-FTM (Web Service-Fault Tolerance Mechanism), which applies this well proven technique to the domain of Web Services to increase system dependability. WS-FTM achieves transparent usage of replicated Web Services by use of a modified stub. The stub is created using tools included in WS-FTM. Our initial implementation includes a simple consensus voter that allows generic result comparison. Finally we show, through the use of a non-trivial example, that WS-FTM can be used to increase the reliability of a Web Service system.

1. Introduction

Web Services are an important component of many Business-to-Business (B2B) systems. Failures generating either downtime or invalid results in such systems can have significant monetary penalties so providing highly reliable systems is an important goal.

The N-Version model is a well proven mechanism for providing protection against physical failure [2] as well as Byzantine faults [1] under certain conditions. Our method applies this technique to the domain of Web Services to achieve increases in system dependability [3].

Our work builds on the work done for FT-Grid [4] which provides a comprehensive system that allows the dynamic construction and execution of Web Services in an n-Version model. The intension is also to provide more complex voting schemes to provide users with more flexibility.

Our purpose is to provide a transparent and easy to use method and tools that will deliver significant increases in dependability whilst at the same time have a minimal impact on system complexity. Given this, we have implemented a much simpler static system with the aim of making it robust and simple to use. It is felt that this will be adequate for most users needs.

In this paper we detail the design and implementation of our fault tolerance method and tools and demonstrate its usefulness via a typical usage scenario. In future work we will address some of the problems associated with common-mode failure by building upon the framework given by WS-FTM.

2. Fault Tolerance

B2B systems are increasingly expected to exhibit highly reliable characteristics particularly in the areas of availability and integrity. When considering service-based applications, many B2B 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 Oriented-Architecture (SOA) may be higher than that for normal applications [5] whilst the heterogeneous nature of services within a SOA means that many service-based applications will be functioning in environments where interaction faults are more likely to occur.

2.1. Redundancy

Redundancy has long been used as a means of increasing the availability of distributed systems, with key components being replicated to protected against hardware failure [3]. Redundancy can be achieved either through: 1) hardware modules can be replicated to provide backup capacity when a failure occurs, 2) using software solutions to replicate key elements of a business process.

Redundancy has many attractions for providing fault tolerance in a system but it also incurs a number of overheads. These include: 1) the cost of replicating calls to replicas; 2) an increase in the complexity of system design and; 3) the cost of providing and maintaining the replicas.

Once redundancy has been introduced into a system it can also be used to protect against Byzantine faults through the use of the n-Version model [6]. Most hardware failures result from either physical defects sustained during manufacturing or development over time as components wear out. This is not the case with software failures since software does not wear out. Software failures result from the invocation of paths that contain faults. Since software is typically more complex than hardware it can be expected to contain many more faults resulting in more failures.

The n-Version model is a design pattern for implementing software fault tolerance. Physical faults, such as machine failure, can be handled by redundancy. Simply replicating a single software component *n* times may not eliminate a particular failure since the software fault will exist in each replica. The n-Version model uses *n* independently implemented versions of a software component run in parallel. By running the components in parallel with the same input data a set of results is obtained. By using a voting mechanism on these results individual failures in a component can be eliminated and the integrity of the final result is increased. The voter guarantees, to some agreed level of integrity, to return a correct result or flag an error.

Since each version of the component is independently implemented each should contain a different set of faults and the voting process can cancel these out. In practice common-mode failures can still occur since developers tend to implement similar solutions to problems [7]. Also Web Service composition can include common Web Services increasing the chance of common-mode failure but there is still a significant gain in reliability [3]. As stated by Knight et al in [7] *"... our result does not mean that N-version programming does not work or should never be used. It means that the reliability of an N-version system may not be as high as theory predicts under the assumption of independence. If the implementation issues can be resolved for a particular N version system, the required reliability might be achieved by using a larger value for N..."*

The voting mechanism can become a single point of failure. This is traditionally overcome by making the voting mechanism simple and conducting extensive testing to determine its reliability. A tiered voting mechanism can be used to reduce the chance of failure through machine failures.

An n-Version model has traditionally been more expensive to design and implement than a single system since each version must be designed, implemented and maintained. With the advent of SOA and Web Services this cost may be reduced since many independent vendors may offer functionally-equivalent Web Services which can be used to construct an *n*-Version system, thus eliminating a large part of the implementation and maintenance cost.

3. WS-FTM

WS-FTM (Web Service-Fault Tolerance Mechanism) is an implementation of the classic n-Version model for use with Web Services. The initial proto-type is implemented using Apache Axis 1.1 for Java. It leverages the standard interface definitions defined by WSDL to create a custom SOAP stub which is used to transparently convert an Axis Service call into a WS-FTM call, vote on the result and supply the result to the client program as a method return.

We envisage WS-FTM being used as both a transparent replication layer to eliminate physical faults caused by machine failures and also as an n-Version model implementation.

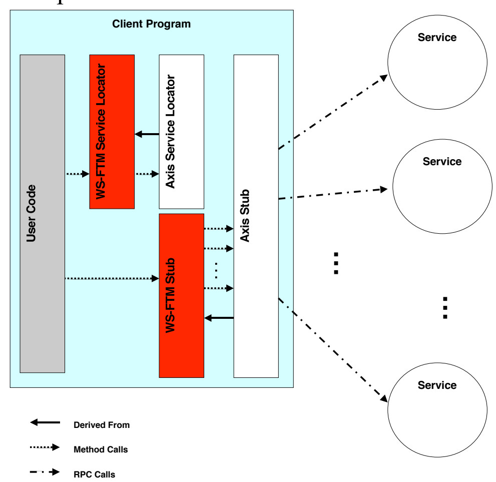

Figure 1: WS-FTM Architecture

3.1. Architecture

WS-FTM uses a simple layered architecture that extends the standard Axis classes. In a standard Axis program the client must first create a *Stub* via the *ServiceLocator* class for the particular interface. The *ServiceLocator* returns a *Stub* that is typecast to the interface defined for the service. These Java classes are generated automatically from the WSDL via the open source WSDL2Java tool supplied with Axis. WS-FTM leverages the simple Axis architecture to implement its voter mechanism. Figure 1 shows the layered architecture used to implement the voter. Replacement *ServiceLocator* and *Stub* classes are generated using the tool WSDL2WSFTM that is part of WS-FTM. These are used in place of the ones generated by WSDL2Java but define the same interface.

The replacement *ServiceLocator* class extends the existing Axis *ServiceLocator* class to supply a new get method that returns a replacement *Stub* rather than an Axis *Stub*. The replacement *Stub* is derived from the Axis *Stub* but implements some extra functionality. It implements a new method (*setServices*) that lets the client supply a list of functionally-equivalent services which define the interface described in the WSDL. These services define the n-Versions required for the voter algorithm.

A new stub method is created for each operation defined in the WSDL in the same way as the Axis *Stub*. These methods implement simple code to execute the service method calls in parallel and then feed the results to the voter. Each service method call is executed using a separate thread to allow concurrent execution. The result computed by the voter is then returned to the client program in the normal way.

The voter implemented by WS-FTM is a simple majority-voting scheme. This scheme was chosen because it is easy to implement and debug. Since the voter is a vital part of the architecture this approach was taken over supplying a more complex voter since the simplicity of the implementation makes it easy to demonstrate its correct operation and thus ensure the integrity of the results. The voter leverages the Java *Comparable* interface to allow generic comparison of results. The *Comparable* interface defines a standard mechanism for comparing objects and is implemented for all the standard Java base objects. By using this mechanism we reduce the complexity of the voter further whilst increasing the range of objects that can be used with it.

This architecture can be easily added to both new code and existing code, since the only changes required are the use of the replacement *ServiceLocator* and supplying the list of functionally-equivalent services to the stub. This comprises two lines of code.

4. Test Case

Any fault tolerance mechanism must demonstrate that it is dependable and does not introduce any unacceptable overheads into a system. The following test cases are used to demonstrate not only the use of WS-FTM in a real world scenario but also that it functions according to these constraints. Our test cases were performed using a network of 2Ghz Pentium 4 machines running Fedora Core 3 with Tomcat 5.5.4, Axis 1.1 and Java 1.4.2 installed. Deployment was via a standard WAR file to the Tomcat app server. Our test code was specifically constructed for this experiment and is available from the project web site.

4.1. Test Scenario

To provide a test bed to demonstrate WS-FTM we have constructed an SOA that simulates a typical stock market trading system (See Figure 2). The 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; and 3) A bank service. The trading service implements a simple automatic buying and selling mechanism. Upper and lower limits are set which trigger trading of shares. Shares are sold when the high limit is exceeded and purchased when the quoted price is less than the lower limit.

We use two configurations to demonstrate WS-FTM. The first is a system implemented without WS-FTM (See Figure 2). This is used to collect a baseline set of measurements so that comparisons can be made. The second is the same system but with the Stock service replicated and called transparently using WS-FTM (See Figure 3).

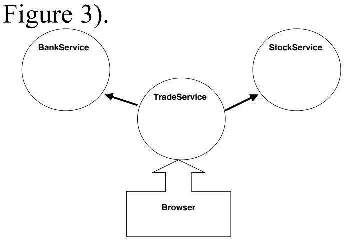

Figure 2: Simple Stock Trading System

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). Since this involves processing and network transfer time a race condition exists which may cause a mismatch between the trigger price and the transaction price. This is intentional and is used to demonstrate the non-invasiveness of WS-FTM since it is possible for incorrected/late stock quote information to cause the system to fail in a noticeable way and makes the system sensitive to introduced latency.

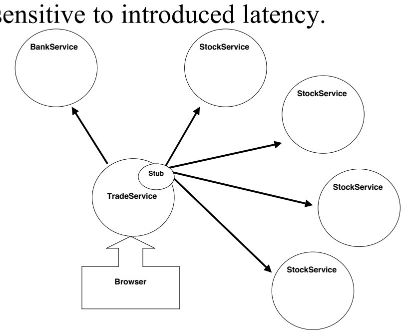

Figure 3: Trading System with Replication

4.2. Overhead Measurement

To measure the overhead introduced by WS-FTM we measured the total time taken to complete an RPC. This is measured from entry into the RPC call from the client code to the result being returned to the client code so for configurations using replication this included the overhead of WS-FTM. We applied this to the non-replicated configuration and then to two different replicated configurations. All Stock services are executed on identical servers so that overheads can be measured. Results are given in Figure 4.

	Non-Replicated	4 Replicas	7 Replicas
Average	14	61	91
Stdev	4	12	45
Sample Size	113	105	108

Figure 4: Overhead (milli-seconds)

Tests were executed over a period of 1 hour per configuration. Variations in RPC exchange sample size over this period are due to variations in client/server logic when failures are reported as no consensus values and hence a slightly different control path is taken.

From this we can see that there is a small but acceptable overhead introduced into the system by WS-FTM. Since all calls to replicated services are executed in separate Java threads the slight increase in overhead can be attributed to the extra processing required to start and process the threads, with the increasing standard deviation being caused by the variation in the time taken for the last consensus value to be returned.

4.3. Corrupted Results

To determine whether WS-FTM increases the reliability of a system, faults were injected into the Web Services returning stock quotes to the trading system. This was done using code insertion [8] with a perturbation function being manually inserted into each Stock service so that each quote returned had a one in five chance of returning a corrupt result. A random number generator controlled the fault injection automatically so that over time the same distribution of faults would be injected into all Stock services.

Four different system configurations were used. The first two use no replication and the second two use replications to increase reliability. The results are given in Figure 5. Each configuration was executed for a period of approximately 24 hours.

	No Replicas No Faults	No Replicas Injected Faults	4 Replicas Injected Faults	7 Replicas Injected Faults
Incorrect Results (%)	0.0	20.8	2.8	0.5
Quote Mismatch (%)	0.0	47.0	12.6	2.5
No-Consensus (%)	*N/A*	*N/A*	15.9	14.1
Sample Size	2337	2451	2551	2265

Figure 5: Corrupted Results

The first column describes the operation of the system with no replication and no injected faults. It is included as a baseline. As can be seen under normal conditions the system functions with no incorrect quotes being returned and no race condition mismatches. Incorrect results and No-Consensus calculations are taken as an average of all quotes whilst mismatches are taken as an average of trigger quotes since a mismatch can only occur on a trigger.

When faults are introduced into the un-replicated system we see a 20% increase in incorrect results which is expected because we have no mechanism in place to prevent them being processed. The 47% mismatches are caused by incorrect quotes being returned as either the trigger or buy/sell price.

The system was then tested using a WS-FTM system using 4 replicas (which should compensate for 1 corrupted service) and then 7 replicas (which should compensate for 2 corrupted services). The results showed a marked improvement in reliability with incorrect results dropping to below 3% when 4 replicas were used. The results also show that a percentage of the attempted requests are rejected since a consensus cannot be reached. This is preferable to returning an incorrect result. There are still a number of mismatch results generated but with 7 replicas this drops to below 3% indicating that the mismatches are caused by the incorrectly returned results and no-consensus errors rather than latency introduced by WS-FTM.

5. Conclusions and Future Work

This paper has detailed our fault tolerance package for use with Web Services and we have demonstrated its operation. We have demonstrated that it: 1) can be easily applied to systems with a minimum of change; and 2) introduces an acceptable amount of overhead into the system demonstrated.

The n-Voter model is a well-understood design pattern that it gives tangible benefits in terms of data integrity and system reliability. This paper demonstrates how WS-FTM can be used to apply it transparently to Web Services.

Future work will include expanding our range of test cases to include more complex fault tolerant scenarios so that WS-FTM can be more comprehensively tested. We will also expand the capabilities of the WSDL2WSFTM tool so that it can handle a greater variety of types. Scalability still needs to be analysed in detail. The algorithm used by the consensus voter attempts to limit latency introduced but it has yet to be proved on a large system. Lastly we intend to use WS-FTM to provide a basis for future work in examining the problem of common-mode failure by eliminating common Web Services/servers in Web Service composition.

6. References

[1] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382-401, 1982.

[2] M. R. Lyu, *Software Fault Tolerance*. Chichester ; New York: John Wiley, 1995.

[3] B. Littlewood, P. Popov, and L. Strigini, "Choosing between Fault-Tolerance and Increased V&V for Improving Reliability," presented at DSN, New York, USA, pp. B42-B43, 2000.

[4] P. Townend and J. Xu, "Replication-based Fault Tolerance in a Grid Environment," presented at U.K. e-Science 3rd All-Hands Meeting, Nottingham, UK, 2004.

[5] C. Lee, "Grid RPC, Events and Messaging," Global Grid Forum Advanced Programming Models Research Group Whitepaper,, September 2001.

[6] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the Association of Computing Machinery*, vol. 27, pp. 228-234, 1980.

[7] J. C. Knight and N. G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," *IEEE Transactions on Software Engineering*, vol. 12, pp. 96-109, 1986.

[8] J. Voas, "Fault Injection for the Masses," *Computer*, vol. 30, pp. 129-130, 1997.