# Making Services Fault Tolerant

Pat. P.W. Chan[1], Michael R. Lyu[1], and Miroslaw Malek[2]

[1] Department of Computer Science and Engineering
The Chinese University of Hong Kong
Hong Kong, China
{pwchan, lyu}@cse.cuhk.edu.hk
[2] Department of Computer Science and Engineering
Humboldt University Berlin, Germany
malek@informatik.hu-berlin.de

**Abstract.** With ever growing use of Internet, Web services become increasingly popular and their growth rate surpasses even the most optimistic predictions. Services are self-descriptive, self-contained, platform-independent and openly-available components that interact over the network. They are written strictly according to open specifications and/or standards and provide important and often critical functions for many business-to-business systems. Failures causing either service downtime or producing invalid results in such systems may range from a mere inconvenience to significant monetary penalties or even loss of human lives. In applications where sensing and control of machines and other devices take place via services, making the services highly dependable is one of main critical goals. Currently, there is no experimental investigation to evaluate the reliability and availability of Web services systems. In this paper, we identify parameters impacting the Web services dependability, describe the methods of dependability enhancement by redundancy in space and redundancy in time and perform a series of experiments to evaluate the availability of Web services. To increase the availability of the Web service, we use several replication schemes and compare them with a single service. The Web services are coordinated by a replication manager. The replication algorithm and the detailed system configuration are described in this paper.

**Keywords:** Web service, availability, redundancy, reliability.

## 1 Introduction

As the use of Web services is growing, there is an increasing demand for dependability. Service-oriented Architectures (SOA) are based on a simple model of roles. Every service may assume one or more roles such as being a service provider, a broker or a user (requestor).

The use of services, especially Web services, became a common practice. In Web services, standard communication protocols and simple broker-request architectures are needed to facilitate an exchange (trade) of services, and this

D. Penkler, M. Reitenspiess, and F. Tam (Eds.): ISAS 2006, LNCS 4328, pp. 43–61, 2006.

model simplifies interoperability. In the coming years, services are expected to dominate software industry. As services begin to permeate all aspects of human society, the problems of service dependability, security and timeliness are becoming critical, and appropriate solutions need to be made available.

Several fault tolerance approaches have been proposed for Web services in the literature [1,2,3,4,5,6,7,8], but the field still requires theoretical foundations, appropriate models, effective design paradigms, practical implementations, and in-depth experimentations for building highly-dependable Web services.

In this paper, related work on dependable services is presented in Section 2, in which the problem statement about reliable Web services is presented. In Section 3, a methodology for reliable Web services is described, in which we propose experimental settings and offer a roadmap to dependable Web services. Experimental results and reliability modeling are presented in Sections 4 and 5. Finally, conclusions are made in Section 6.

## 2 Related Work

It is a well-known fact that fault tolerance can be achieved via spatial or temporal redundancy, including replication of hardware (with additional components), software (with special programs), and time (with the diversified of operations) [9,10,11].

Spatial redundancy can be dynamic or static. Both use replication but in static redundancy, all replicas are active at the same time and voting takes place to obtain a correct result. The number of replicas is usually odd and the approach is known as n-modular redundancy (NMR). For example, under a single-fault assumption, if services are triplicated and one of them fails, the remaining two will still guarantee the correct result. The associated spatial redundancy cost is high (i.e., three copies plus a voter). The time overhead of managing redundant modules such as voting and synchronization is also considerably large for static redundancy. Dynamic redundancy, on the other hand, engages one active replica at one time while others are kept in an active or in a standby state. If one replica fails, another replica can be employed immediately with little impact on response time. In the second case, if the active replica fails, a previously inactive replica must be initialized and take over the operations. Although this approach may be more flexible and less expensive than static redundancy, its cost may still be high due to the possibility of hastily eliminating modules with transient faults. It may also increase the recovery time because of its dependence on time-consuming error-handling stages such as fault diagnosis, system reconfiguration, and resumption of execution. Redundancy can be achieved by replicating hardware modules to provide backup capacity when a failure occurs, or redundancy can be obtained using software solutions to replicate key elements of a business process.

In any redundant systems, common-mode failures (CMFs) result from failures that affect more than one module at the same time, generally due to a common cause. These include design mistakes and operational failures that may be caused

externally or internally. Design diversity has been proposed in the past to protect redundant systems against common-mode failures [12,13] and has been used in both firmware and software systems [14,15,16]. The basic idea is that, with different design and implementations, common failure modes can be reduced. In the event that they exist and are manifested, they will probably cause different failure effects. One of the design diversity techniques is N-version programming, and another one is Recovery Blocks. The key element of N-version programming or Recovery Blocks approaches is diversity. By attempting to make the development processes diverse it is anticipated that the independently designed versions will also contain diverse faults. It is assumed that such diverse faults, when carefully managed, will minimize the likelihood of coincident failures.

Based on the discussed techniques, a number of reliable Web services techniques appeared in the recent literature. A Web Services Reliable Messaging Protocol is proposed in [1], which employs flooding and acknowledgement to ensure that messages are delivered reliably among distributed applications, in the presence of software component, system, or network failures.

WS-FTM (Web Service-Fault Tolerance Mechanism) is an implementation of the classic N-version model with Web services [2] which can easily be applied to systems with a minimal change. The Web services are implemented in different versions and the voting mechanism is in the client program.

FT-SOAP [3], on the other hand, is aimed at improving the reliability of the SOAP when using Web services. The system includes different function replication management, fault management, logging/recovery mechanism and client fault tolerance transparency. FT-SOAP is based on the work of FT-CORBA [4], in which a fault-tolerant SOAP-based middleware platform is proposed. There are two major targets in FT-SOAP: 1) to define a fault-tolerant SOAP service standard recommendation, and 2) to implement an FT-SOAP service prototype.

FT-Grid [5] is another design, which is a development of design-diverse fault tolerance in Grid. It is not specified for Web services but the techniques are applicable to Web services. The FT-Grid allows a user to manually search through any number of public or private UDDI repositories, select a number of functionally-equivalent services, choose the parameters to supply to each service, and invoke those services. The application can then perform voting on the results returned by the services, with the aim of filtering out any anomalous results.

Although a number of approaches have been proposed to increase the Web service reliability, there is a need for systematic modeling and experiments to understand the tradeoffs and verify reliability of the proposed methods.

## 3 Problem Statement

There are many fault-tolerant techniques that can be applied to Web services including replication and diversity. Replication is one of the efficient ways for creating reliable systems by time or space redundancy. Redundancy has long been used as a means of increasing the availability of distributed systems, with

key components being re-executed (replication in time) or replicated (replication in space) to protect against hardware malfunctions or transient system faults. Another efficient technique is design diversity. By independently designing software systems or services with different programming teams, diversity provides an ultimate resort in defending against permanent software design faults.

In this paper, we focus on the systematic analysis of the replication techniques when applied to Web services. We analyze the performance and the availability of the Web services using spatial and temporal redundancy and study the tradeoffs between them. A generic Web service system with spatial as well as temporal replication is proposed and experimented with.

## 4 Methodologies for Reliable Web Services

### 4.1 Failure Response Stages of Web Services

Web services go through different operation modes, so when failures occur and the failure response of Web services can be classified into different stages [17]. When a failure occurs, the Web service should confine the failure by applying fault detection techniques to find out the failure causes and the failed components should be repaired or recovered. Then, reconfiguration, restart and reintegration should follow. The flow of the failure response of a Web service is shown in Figure 1 and the details of each stage are described as follows:

**Fault confinement.** This stage limits the fault impact by attempting to contain the spread of fault effects in one area of the Web service, thus preventing contamination of other areas. Fault-confinement can be achieved through the

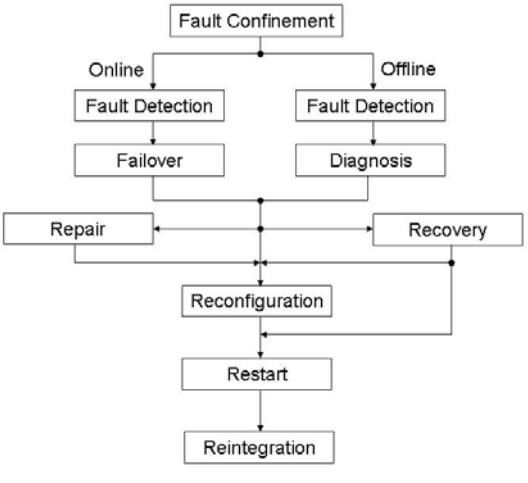


**Fig. 1.** Flow of the failure response of Web services

use of fault detection within the Web services, consistency checks, and multiple requests/confirmations.

**Fault detection.** This stage recognizes that something unexpected has occurred in a Web service. Fault latency is the period of time between the occurrence of a fault and its detection. Techniques fall here into two classes: off-line and on-line. With off-line techniques, such as diagnostic programs, the service is not able to perform useful work while under test. On-line techniques, such as duplication, provide a real-time detection capability that is performed concurrently with useful work.

**Diagnosis.** This stage is necessary if the fault detection technique does not provide information about the fault location. Typically, fault diagnosis encompasses both fault detection and fault location.

**Reconfiguration.** This stage occurs when a fault is detected and located. The Web service can be composed of different components. When providing the service, there may be a fault in individual components. The system may reconfigure its components either to replace the failed component or to isolate it from the rest of the system.

**Recovery.** This stage utilizes techniques to eliminate the effects of faults. Three basic recovery approaches are available: fault masking, retry and rollback. Fault-masking techniques hide the effects of failures by allowing redundant information to outweigh the incorrect information. Web services can be replicated or implemented with different versions (NVP). Retry undertakes a second attempt at an operation and is based on the premise that many faults are transient in nature. Web services provide services through a network, and retry would be a practical approach as requests/replies may be affected by the state of the network. Rollback makes use of the fact that the Web service operation is backed up (checkpointed) at some point in its processing prior to fault detection and operation recommences from that point. Fault latency is important here because the rollback must go back far enough to avoid the effects of undetected errors that occurred before the detected error.

**Restart.** This stage occurs after the recovery of undamaged information.

- Hot restart: resumption of all operations from the point of fault detection and is possible only if no damage has occurred.
- Warm restart: only some of the processes can be resumed without loss.
- Cold restart: complete reload of the system with no processes surviving. The Web services can be restarted by rebooting the server.

**Repair.** At this stage, a failed component is replaced. Repair can be off-line or on-line. Web services can be component-based and consist of other Web services. In off-line repair, either the Web service will continue if the failed component/sub-Web service is not necessary for operation or the Web

services must be brought down to perform the repair. In on-line repair, the component/sub-Web service may be replaced immediately with a backup spare or operation may continue without the component. With on-line repair, the Web service operation is not interrupted.

**Reintegration.** At this stage the repaired module must be reintegrated into the Web service. For on-line repair, reintegration must be performed without interrupting the Web service operation.

### 4.2 Proposed Technique

In the previous section, we describe a general approach in system fault tolerance which can be applicable to Web services. In the following section, we propose a replication Web service system for reliable Web services. In our system, the dynamic approach is considered and its architecture is shown in Figure 2.

**Scheme details.** In the proposed system, one Web server works as the active server and others are used for backup purpose to tolerate a single server failure. The Web service is replicated on different machines, but only one Web service provides the requested service at a time, which is called the primary Web service. The Web service is replicated identically on different machines; therefore, when the primary Web service fails, the other Web services can immediately provide the required service. The replication mechanism shortens the recovery time and increases the availability of the system.

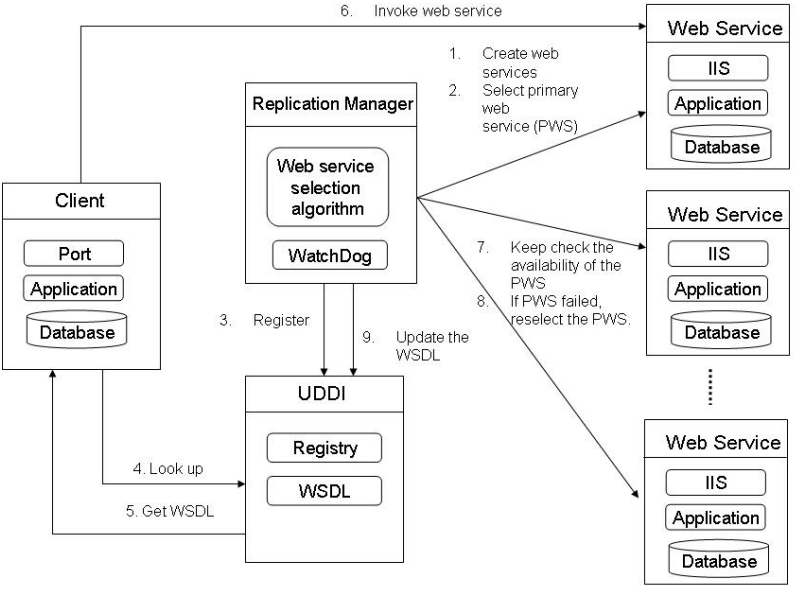


**Fig. 2.** Proposed architecture for dependable Web services

The main component of this system is the replication manager, which acts as a coordinator of the Web services. The replication manager is responsible for:

1. Creating a Web service.
2. Choosing (with anycasting algorithm) the best (fastest, most robust, etc.) Web service [18] to provide the service which is called the primary Web service.
3. Registering the Web Service Definition Language (WSDL) with the Universal Description, Discovery, and Integration (UDDI).
4. Continuously checking the availability of the primary Web service by using a watchdog.
5. Selecting another Web service provider if the primary service fails, so as to ensure fault tolerance.

When the primary Web service fails, the replication manager selects the most suitable Web service again to continue providing the service. The replication manager maps the new address of the new primary Web service to the WSDL, thus the clients can still access the Web service with the same URL. This failover process is transparent to the users. The detailed procedure is shown in Figure 2.

The workflow of the replication manager is shown in Figure 3. The replication manager is running on a server, which keeps checking the availability of the Web services by the polling method. It sends messages to the Web services periodically. If it does not get the reply from the primary Web service, it will select another Web service to replace the primary one and map the new address to the WSDL. The system is considered failed if all the Web services have failed. If a Web service replies with the logging, the replication manager will record the information of the Web service.

### 4.3 Roadmap for Experimental Research

We take a pragmatic approach by starting with a single service without any replication. The only approach to fault tolerance in this case is the use of redundancy in time. If a service is considered as an atomic action or a transaction where the input is clearly defined, no interaction is allowed during its execution and the termination has two outcomes: correct or incorrect. In this case, the only way to make such service fault tolerant is to retry or reboot it. This approach allows tolerance of temporary faults, but it will not be sufficient for tolerating permanent faults within a server or a service. One issue is how much delay can a user tolerate, and another issue is the optimization of the retry or the reboot time; in other words, deciding when a current request should be timed out. By handling services as atomic transactions, exception handling does not help in dealing directly with inherent problems of a service. Consequently, continuous service is only possible by performing re-execution using a retry or reboot at the termination points or after a timeout period.

If redundancy in time is not appropriate to meet dependability requirements or time overhead is unacceptable, the next step is redundancy in space. Redundancy in space for services means replication where multiple copies of a given

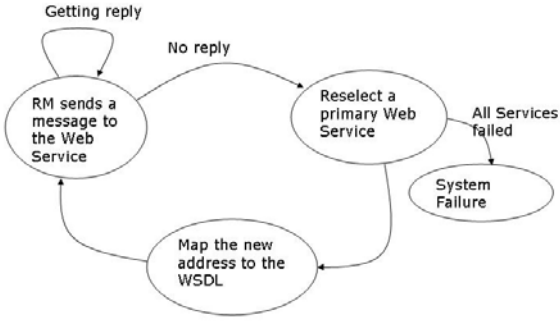


**Fig. 3.** Workflow of the Replication Manager

service may be executed sequentially or in parallel. If the copies of the same services are executed on different servers, different modes of operation are possible:

1. Sequentially, meaning that we await a response from a primary service and in case of timeout or a service delivering incorrect results, we invoke a backup service (multiple backup copies are possible). It is often called failover.
2. In parallel, meaning that multiple services are executed simultaneously and if the primary service fails, the next one takes over [1]. It is also called a failover. Another variant is that the service whose response arrives first is taken.
3. There is also a possibility of majority voting using n-modular redundancy, where results are compared and the final outcome is based on at least $\lfloor n/2 + 1 \rfloor$ services agreeing on the result.
4. If diversified versions of different services are compared, the approach can be seen as either a Recovery Blocks (RB) system where backup services are engaged sequentially until the results are accepted (by an Acceptance Test), or an N-version programming (NVP) system where voting takes place and majority results are taken as the final outcome. In case of failure, the failed service can be masked and the processing can continue.

NVP and RB have undergone various challenges and vivid discussions. Critics would state that the development of multiple versions is too expensive and dependability improvement is questionable in comparison to a single version, provided the development effort equals the development cost of the multiple versions. We argue that in the age of service-oriented computing, diversified Web services permeate and the objections to NVP or RB can be mitigated. Based on market needs, service providers competitively and independently develop their services and make them available to the market. With abundance of services for specific functional requirements, it is apparent that fault tolerance by design

[1] In such case service parameter compatibility must be checked or aligned. Services are assumed to have equivalent functionality.

diversity will be a natural choice. NVP should be applied to services not only for dependability but also for higher performance purpose.

Finally, a hybrid method may be used where both space and time redundancy are applied, and depending on system parameters, a retry might be more effective before switching to the backup service. This type of approach will require a further investigation.

We also need to formulate several additional quality-of-service parameters to service customers. We propose a number of fault injection experiments showing both dependability and performance with and without diversified Web services. The outlined roadmap to fault-tolerant services leads to ultra reliable services where hybrid techniques of spatial and time redundancy can be employed for optimizing cost-effectiveness tradeoffs. In the next section, we describe the various approaches and some experiments in more detail.

### 4.4 Experiments

A series of experiments are designed and performed for evaluating the reliability of the Web service, including single service without replication, single service with retry or reboot, and a service with spatial replication. We will also perform retry or failover when the Web service is down. A summary of five (1-5) experiments is stated in Table 1.

**Table 1.** Summary of the experiments

| | | None | Retry/Reboot | Failover | Both(hybrid) |
|---|---|---|---|---|---|
| 1 | Single service, no retry | 1 | – | – | – |
| 2 | Single service with retry | – | 2 | – | – |
| 3 | Single service with reboot | – | 3 | – | – |
| 4 | Spatial replication | – | – | 4 | – |
| 5 | Spatial replication | – | – | – | 5 |

Our experimental system is implemented with Visual Studio .Net and runs with .Net framework. The Web service is replicated on different machines and the primary Web service is chosen by the replication manager.

In the experiments, faults are injected in the system and the fault injection techniques are similar, for example, to the ones referred in [6,22]. A number of faults may occur in the Web service environment [20,21], including network problem, resource problem, entry point failure, and component failure. These faults are injected in the experimental system to evaluate the reliability of our proposed scheme. Our experimental environment is defined by a set of parameters. Table 2 shows the parameters of the Web service in our experiments.

**Experimental results.** We compare five approaches for providing the Web services. The details of the experiments are described as follows:

**Table 2.** Parameters of the experiments

| | Parameters | Current setting/metric |
|---|---|---|
| 1 | Request frequency | 1 req/min |
| 2 | Polling frequency | 10 per min |
| 3 | Number of replicas | 5 |
| 4 | Client timeout period for retry | 10 s |
| 5 | Failure rate $\lambda$ | number of failures/hour |
| 6 | Load (profile of the program) | percentage or load function |
| 7 | Reboot time | 10 min |
| 8 | Failover time | 1 s |

1. **Single service without retry and reboot.** The Web service is provided by a single server without any replication. No redundancy technique is applied to this Web service.
2. **Single service with retry.** The Web service provides the service and the client retries another Web service when there is no response from the original Web service after timeout.
3. **Single service with reboot (restart).** The Web service provides the service and the Web service server will reboot when there is no response from the Web service. Clients will not retry after timeout when there is no response from the service.
4. **Spatial replication with failover.** We use a generic spatial replication: The Web service is replicated on different machines and the request is transferred to another machine when the primary Web service fails (failover). The replication manager coordinates among the replicas and carries out a failover in case of a failure. Clients will only submit the request once and will not retry.
5. **Spatial replication with failover and retry.** This is a hybrid approach. Similar to the Experiment 4 where the Web service is replicated on different machines and the request is transferred to another one (failover) when the primary Web service fails. But the client will retry if there is no response from the Web service after timeout.

The Web services were executed for 720 hours generating a total of 720x60 req/hr = 43200 requests from the client. A single failure is counted when the system cannot reply to the client. For the approach with retry, a single failure is counted when a client retries five times and still cannot get the result. A summary of the results is shown in Table 3 and the Figures 4 to 7 depict the number of failures as the time increases.

The reliability of Web services is tested under different scenarios, including normal operation, resource problem by increasing the load of the server, entry point failure by rebooting the server periodically, and a number of faults by fault injection techniques.

In the fault injection, WS-FIT fault injection is applied. The WS-FIT fault injection method is a modified of Network Level Fault Injection. WS-FIT differs from

**Table 3.** Experimental results

| Experiments over 720 hour period (43200 reqs) | Normal | Resource Problem | Entry Point Failure | Network Level Fault Injection |
|---|---|---|---|---|
| Exp1 | 4928 | 6130 | 6492 | 5324 |
| Exp2 | 2210 | 2327 | 2658 | 2289 |
| Exp3 | 2561 | 3160 | 3323 | 5211 |
| Exp4 | 1324 | 1711 | 1658 | 5258 |
| Exp5 | 1089 | 1148 | 1325 | 2210 |

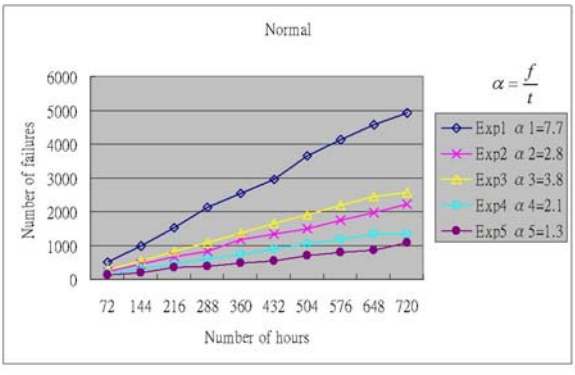


**Fig. 4.** Number of failures when the server operates normally

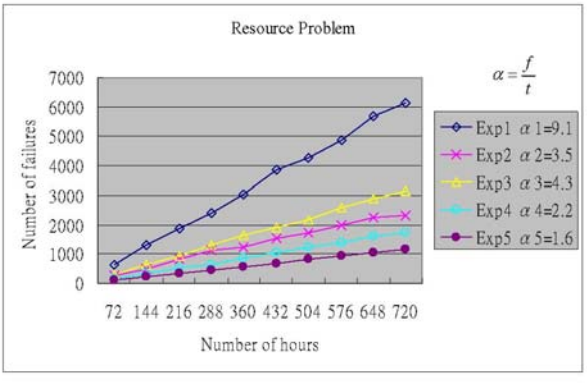


**Fig. 5.** Number of failures under resource problem

standard Network Level Fault Injection techniques in that the fault injector decodes the SOAP message and injects faults into individual remote procedure call (RPC) parameters, rather than randomly corrupting a message, for instance by bit-flipping. This enables API-level parameter value modification to be performed in a non-invasive way as well as the standard Network Level Fault Injection.

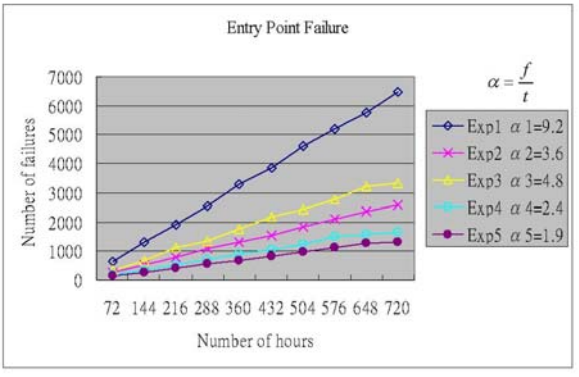


**Fig. 6.** Number of failures under entry point failure

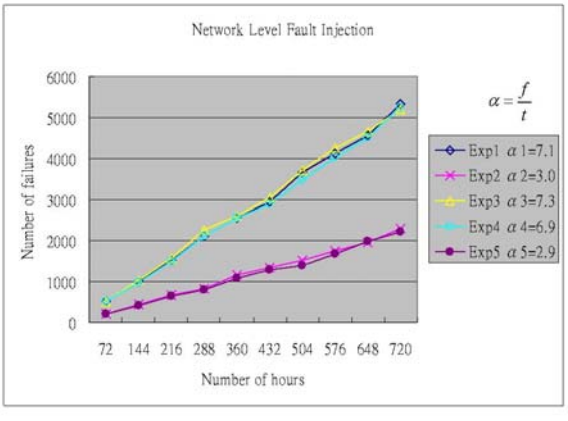


**Fig. 7.** Number of failures under Network Level Fault Injection

**Discussion.** The experiments barely indicate rather linear increase of the number of failures in all categories. Therefore, the ratio of the number of failures to the amount of time is almost constant for each type of experiments and each type of failures. The ratio $\alpha$ (number of failures divided by time $t$) is given for each type of experiments and each type of failures in Figures 4 to 7. So the communicative number of failures $f$ at time $t$ can be approximate by $f = \alpha \times t$.

When there is no redundancy techniques applied on the Web service system (Exp 1), it is clearly shown that the failure rate of the system is the highest. Consequently, we try to improve the reliability of the Web service in two different ways, including spatial redundancy with replication and temporal redundancy with retry or reboot.

*Resource Problem and Entry Point Failure.* Under different situations, our approach improves the availability of the system differently. When the system is

under resource problem and entry point failure, the experiment shows that the temporal redundancy helps to improve the availability of the system.

For the Web service with retry (Exp 2), the percentage of failures of the system (number of failed requests/total number of requests) is reduced from 11.97% to 4.93%. This shows that the temporal redundancy with retry can significantly improve the availability of the Web service. When there is a failure occurrence in the Web service, on the average, the clients need to retry twice to get the response from the Web service.

Another temporal redundancy is Web service with reboot (Exp 3). From the experimental result, it is found that the failure rate of the system is also reduced: from 11.97% to 6.44%. The improvement is good, but not as substantial as the temporal redundancy with retry. It is due to the fact that when the Web service cannot be provided, the server will take time to reboot.

Spatial redundancy is applied in the system in Exp 4, which is the approach we have proposed. The availability of the system is improved even more significantly: the failure rate of the system is reduced from 11.97% to 3.56%. The Web service performs failover when the Web service cannot respond. The replication manager keeps checking the availability of the Web services. If the primary service fails, the replication manager selects another Web service to provide the service. The replication manager sends a message to the Web server to check the availability every 5 ms. It shortens the potential downtime of the Web service, thus the failure rate is reduced. In the experiment, on the average, the replication manager detects that there are around 600 failures in the Web services and performs the failovers accordingly.

To further improve the reliability of the Web service, both spatial and temporal redundancy is applied in the system in the Experiment 5. The failure rate is reduced from 11.97% to 2.59%. In the experiment, the Web service is replicated on five different machines and the clients will retry if they cannot get response correctly from the service. It is demonstrated that this setting results in the lowest failure rate. This shows that spatial and temporal redundancy (a hybrid approach) achieve the highest gain in reliability improvement of the Web service.

*Network Level Fault Injection.* When the system is under network level fault injection, the temporal redundancy reduces the failure rate of the system from 12.32% to 5.12%. When there are fault injected into the SOAP message, the system cannot process the request correctly, which will cause error in the system. However, with temporal redundancy, the clients can resubmit the result to the system when there is a fault injected into the previous message; thus, the failure rate of the system is reduced. However, the spatial redundancy approach cannot improve the availability of the system. It is because even the message has injected faults and it will not trigger a failover of the system.

*Failure Rate.* The failure rate of a system is defined as:

$$\lambda = \lim_{\Delta \longrightarrow 0} \frac{F(t + \Delta t) - F(t)}{\Delta t} \tag{1}$$

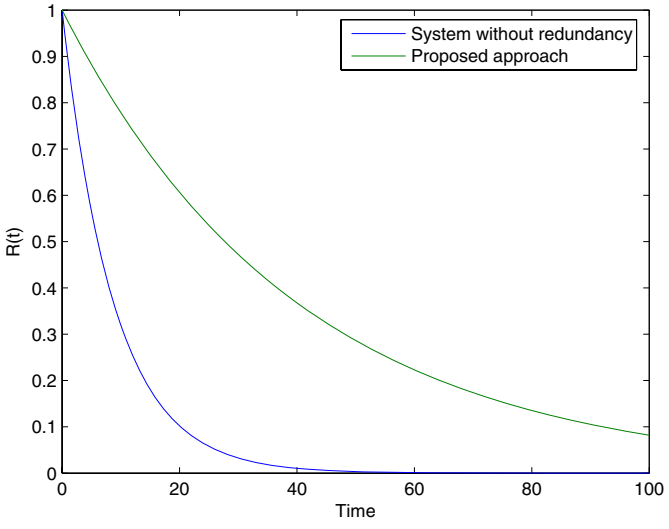


**Fig. 8.** Reliability of the system over time

The failure rate of our system, using a specific scenario, has improved from 0.114 to 0.025. The reliability of the system can be calculated with

$$R(t) = e^{-\lambda(t)t} \tag{2}$$

and Figure 8 shows the reliability of the discussed system.

Availability of the system is defined as:

$$A = \frac{MTTF}{MTTF + MTTR} \tag{3}$$

And in our case

$$\begin{aligned} MTTF &= \frac{1}{\lambda(t)} \\ &= \frac{1}{0.025} = 40 \\ MTTR &= 3s \end{aligned}$$

$$A = \frac{40}{40+3} = 0.93$$

which is quite of an improvement from $A = 0.75$ but still not up to standards of today's expectations.

## 5 Reliability Modeling

We develop the reliability model of the proposed Web service paradigm using Markov chains [23]. The model is shown in Figure 9. The reliability model is analyzed and verified through applying the reliability evaluation tool SHARPE [25].

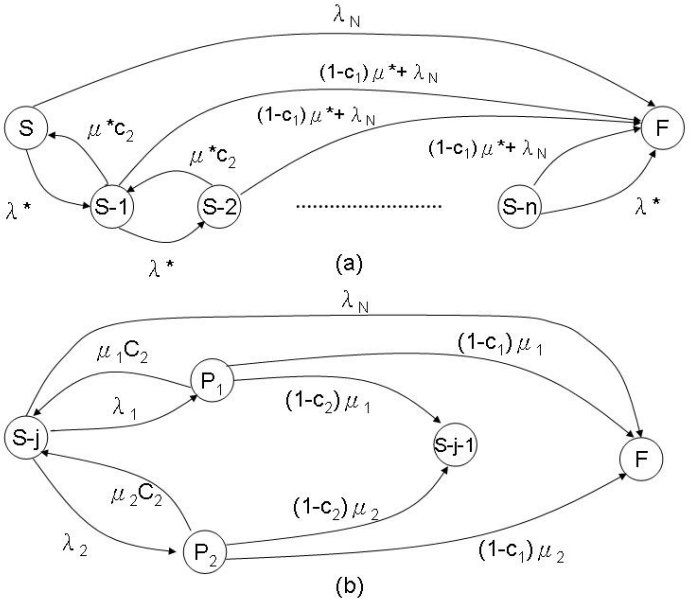


**Fig. 9.** Markov chain based reliability model for the proposed system

In Figure 9(a), the state $s$ represents the normal execution state of the system with $n$ Web service replicas. In the event of an error, the primary Web service fails, and the system will either go into the other states (i.e., $s-j$ which represents the system with $n-j$ working replicas remaining, if the replication manager responds on time), or it will go to the failure state $F$ with conditional probability $(1-c_1)$. $\lambda^*$ denotes the error rate at which recovery cannot complete in this state and $c_1$ represents the probability that the replication manager responds on time to switch to another Web service.

When the failed Web service is repaired, the system will go back to the previous state, $s-j+1$. $\mu^*$ denotes the rate at which successful recovery is performed in this state, and $c_2$ represents the probability that the failed Web service server reboots successfully. If the Web service fails, it switches to another Web service. When all Web services fail, the system enters the failure state $F$. $\lambda_n$ represents the network failure rate.

In Figure 9, $(s-1)$ to $(s-n)$ represent the working states of the $n$ Web service replicas and the reliability model of each Web service is shown in Figure 9(b).

**Table 4.** Model parameters

| ID | Description | Value |
|---|---|---|
| $\lambda_N$ | Network failure rate | 0.01 |
| $\lambda^*$ | Web service failure rate | 0.025 |
| $\lambda_1$ | Resource problem rate | 0.142 |
| $\lambda_2$ | Entry point failure rate | 0.150 |
| $\mu^*$ | Web service repair rate | 0.286 |
| $\mu_1$ | Resource problem repair rate | 0.979 |
| $\mu_2$ | Entry point failure repair rate | 0.979 |
| $C_1$ | Probability that the RM responds on time | 0.9 |
| $C_2$ | Probability that the server reboots successfully | 0.9 |

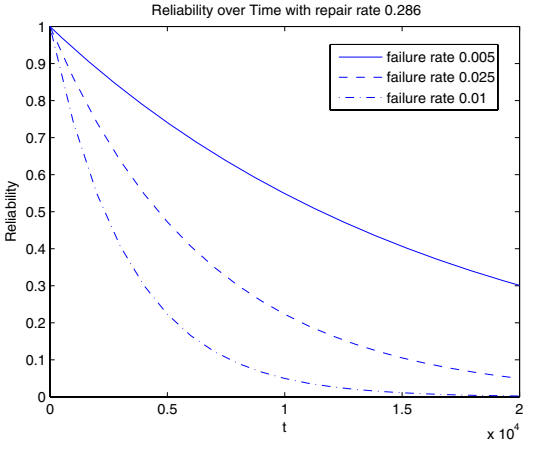


**Fig. 10.** Reliability with different failure rate and repair rate is 0.286

There are two types of failures simulated in our experiments: $P_1$ denotes recourses problem (server busy) and $P_2$ denotes entry point failures (server reboot). If a failure occurs in the Web service, either the Web service can be repaired with $\mu_1$ (to enter $P_1$) or $\mu_2$ (to enter $P_2$) repair rates with conditional probability $c_1$, or the error cannot be recovered, and the system enters the next state $(s - j - 1)$ with one less Web service replica available. If the replication manager cannot respond on time, it will go to the failure state. From the figure, two formulas can be obtained:

$$\lambda^* = \lambda_1 \times (1 - C_1)\mu_1 + \lambda_2 \times (1 - C_2)\mu_2 \tag{4}$$

$$\mu^* = \lambda_1 \times \mu_1 + \lambda_2 \times \mu_2 \tag{5}$$

From the experiments, we obtain the error rates and the repair rates of the system, and the values are shown in Table 4.

According to the parameters obtained from the experiments, the reliability of the system over time is calculated with the tool SHARPE. In Figure 10, the

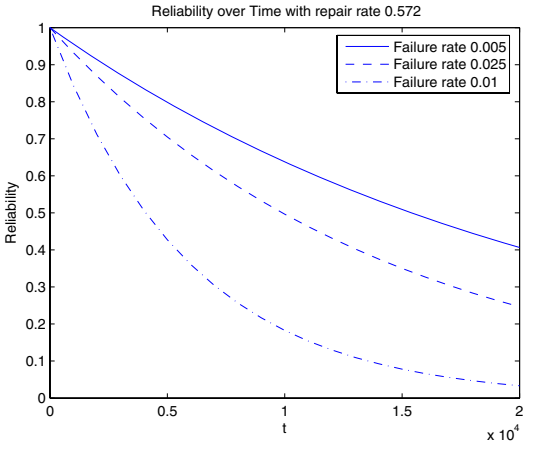


**Fig. 11.** Reliability with different failure rate and repair rate is 0.572

repair rate $\mu^*$ is 0.286 failure/s (from the experiment), the reliability is plotted under different failure rate $\lambda^*$. Note that the failure rate obtained from the experiments is 0.025 failure/s. This failure rate is measured under an accelerated testing environment. By considering the test compression factor [26], the failure rate of the system in a real situation will be much less. A similar reliability curve is plotted in Figure 11 with repair rate $\mu^*$ equal to 0.572 failure/s.

## 6 Conclusions

In the paper, we surveyed and addressed applicability of replication and design diversity techniques for reliable services and proposed a hybrid approach to improving the availability of Web services. Our approach reduces the number of failures in comparison to normal singular method by a factor of about 5. Furthermore, we carried out a series of experiments to evaluate the availability and reliability of the proposed Web service system. From the experiments, we conclude that both temporal and spatial redundancy are important to the availability improvement of the Web service. In the future, we plan to test the proposed schemes with a wide variety of systems, environments and fault injection scenarios and analyze the impact of various parameters on reliability and availability. Moreover, we will evaluate effectiveness of the schemes with design diversity techniques in handling permanent design faults in Web services.

## Acknowledgement

We would like to thank Prof. Kishor Trivedi for providing SHARPE for our reliability analysis. The work described in this paper was fully supported by two grants: One from the Research Grants Council of the Hong Kong Special

Administrative Region, China (Project No. CUHK4205/04E), and another from the Shun Hing Institute of Advanced Engineering (SHIAE) of The Chinese University of Hong Kong.

## References

1. R. Bilorusets, A. Bosworth et al, "Web Services Reliable Messaging Protocol WS-ReliableMessaging," EA, Microsoft, IBM and TIBCO Software, http://msdn.microsoft.com/library/enus/dnglobspec/html/ws-reliablemessaging.asp, Mar. 2004.
2. N. Looker and M. Munro, "WS-FTM: A Fault Tolerance Mechanism for Web Services," University of Durham, Technical Report, 19 Mar. 2005.
3. D. Liang, C. Fang, and C. Chen, "FT-SOAP: A Fault-tolerant Web Service, " Institute of Information Science, Academia Sinica, Technical Report 2003.
4. D. Liang, C. Fang and S. Yuan, "A Fault-Tolerant Object Service on CORBA," Journal of Systems and Software, Vol. 48, pp. 197-211, 1999.
5. P. Townend, P. Groth, N. Looker, and J. Xu, "Ft-grid: A fault-tolerance system for e-science," Proc. of the UK OST e-Science Fourth All Hands Meeting (AHM05), Sept. 2005.
6. M. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan, "Thema: Byzantine-Fault-Tolerant Middleware for Web-Service Application," Proc. of IEEE Symposium on Reliable Distributed Systems, Orlando, FL, Oct. 2005.
7. A. Erradi and P. Maheshwari, "A broker-based approach for improving Web services reliability", Proc. of IEEE International Conference on Web Services, vol. 1, pp. 355-362, 11-15 Jul. 2005.
8. W . Tsai, Z. Cao, Y. Chen, Y and R. Paul, "Web services-based collaborative and cooperative computing," Proc. of Autonomous Decentralized Systems, pp. 552-556, 4-8 Apr. 2005.
9. D. Leu, F. Bastani and E. Leiss, "The effect of statically and dynamically replicated components on system reliability," IEEE Transactions on Reliability, vol.39, Issue 2, pp.209-216, Jun. 1990.
10. B. Kim, "Reliability analysis of real-time controllers with dual-modular temporal redundancy," Proc. of the Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA) 1999, pp.364-371, 13-15 Dec. 1999.
11. K. Shen and M. Xie, "On the increase of system reliability by parallel redundancy," IEEE Transactions on Reliability, vol.39, Issue 5, pp.607-611, Dec. 1990.
12. A. Avizienis, and L. Chen, "On the implementation of N-version programming for software fault-tolerance during program execution," Proc. of First International Computer Software and Applications Conference, pp.149-155, 1977.
13. A. Avizienis, and J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," IEEE Transactions on Computer, pp. 67-80, Aug. 1984.
14. M.R. Lyu and A. Avizienis, "Assuring Design Diversity in N-Version Software: A Design Paradigm for N-Version Programming," in Fault-Tolerant Software Systems: Techniques and Applications, H. Pham (ed.), IEEE Computer Society Press Technology Series, pp. 45-54, Oct. 1992.
15. J. Lala, and R. Harper, "Architectural principles for safety-critical real-time applications," Proc. of the IEEE, vol. 82, no.1, pp.25-40, Jan. 1994.

16. R. Riter, "Modeling and Testing a Critical Fault-Tolerant Multi-Process System," Proc. the 25th International Symposium on Fault-Tolerant Computing, pp.516-521, 1995.
17. M. Lyu and V. Mendiratta, "Software Fault Tolerance in a Clustered Architecture: Techniques and Reliability Modeling," Proc. of 1999 IEEE Aerospace Conference, Snowmass, Colorado, vol.5, pp.141-150, 6-13 Mar. 1999.
18. M. Sayal, Y. Breitbart, P. Scheuermann, and R. Vingralek, "Selection algorithms for replicated web servers," Proc. of Workshop on Internet Server Performance 98, Madison, WI, Jun. 1998.
19. N. Looker, M. Munro, and J. Xu, "Simulating Errors in Web Services," International Journal of Simulation: Systems, Science and Technology, vol.5, pp.29-38, 2004.
20. Y. Yan, Y. Liang and X. Du, "Controlling remote instruments using Web services for online experiment systems," Proc. of IEEE International Conference on Web Services (ICWS) 2005, 11-15 Jul. 2005.
21. Y. Yan, Y. Liang and X. Du, "Distributed and collaborative environment for online experiment system using Web services," Proc. the Ninth International Conference on Computer Supported Cooperative Work in Design 2005, vol.1, pp.265-270, 24-26 May 2005.
22. N. Looker and J. Xu, "Assessing the Dependability of SOAP-RPC-Based Web Services by Fault Injection," Proc. of the 9th IEEE International Workshop on Object-oriented Real-time Dependable Systems, pp.163-170, 2003.
23. K. Goseva-Popstojanova and K. Trivedi, "Failure correlation in software reliability models," IEEE Transactions on Reliability, vol.49, Issue 1, pp.37-48, Mar. 2000.
24. H. Guen, R. Marie and T. Thelin, "Reliability estimation for statistical usage testing using Markov chains," Proc. of the 15th International Symposium on Software Reliability Engineering (ISSRE) 2004, pp.54-65, 2-5 Nov. 2004.
25. R. Sahner, K. Trivedi, and A. Puliafito, "Performance and Reliability Analysis of Computer Systems. An Example-BasedApproach Using the SHARPE Software Package," Kluwer, Boston, MA (1996).
26. M. Lyu, "Handbook of Software Reliability Engineering," IEEE Computer Society Press and McGraw-Hill Book Company.