HAL
open science

Simulation Based Analysis Of Middleware Service Impact On System Reliability: Experiment On Java Application Server

Gang Huang, Weihu Wang, Tiancheng Liu, Hong Mei

To cite this version:

Gang Huang, Weihu Wang, Tiancheng Liu, Hong Mei. Simulation Based Analysis Of Middleware Service Impact On System Reliability: Experiment On Java Application Server. Journal of Systems and Software, 2011, 84 (7), pp.1160-1170. hal-00644654

HAL Id: hal-00644654

https://inria.hal.science/hal-00644654v1

Submitted on 30 Nov 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

SIMULATION-BASED ANALYSIS OF MIDDLEWARE SERVICE IMPACT ON SYSTEM RELIABILITY: EXPERIMENT ON JAVA APPLICATION SERVER

Gang HUANG, Weihu WANG, Tiancheng LIU, Hong MEI

School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, China
Corresponding to huanggang@sei.pku.edu.cn

Being a popular runtime infrastructure in the era of Internet, middleware provides more and more services to support the development, deployment and management of distributed systems. At the same time, the reliability of middleware services has a significant impact on the overall reliability of the system. Different services have different impacts and different service fault-tolerance solutions have different cost and risk. Therefore, the identification of the services that greatly affect the whole system reliability is the major obstacle to achieving reliable middleware-based systems. In this paper, we present an analytical framework to automatically reason and quantify such impacts when deploying the target system. In this framework, faults are represented by exceptions in modern programming languages; service failures are simulated by software fault injection; reliability impacts are measured by scenarios. This framework is demonstrated on multiple JEE application servers, including JBoss, JonAS and PKUAS. The experiments on two JEE blueprint applications, namely JPS and ECperf, show the feasibility, the applicability and the usability of this framework.

Keywords: reliability; middleware service; impact analysis; software implemented fault injection.

1. Introduction

The rapid evolution and pervasiveness of network and distributed applications has led to a proliferation of middleware services. Such a proliferation can be perceived from three dimensions. First, middleware encapsulates more capabilities to manage underlying computing resources while these functions are traditionally considered as the major functions of distributed operating systems. Second, although middleware technologies have been initially developed to address recurrent problems in the development of distributed systems, they often implement additional functions that are only reusable in a specific application domain, such as finances, retails, telecommunications, etc. Third, a middleware provides some additional facilities, like component models and deployment tools, which ease the development and deployment of distributed systems. As a result, modern middleware products provide much more diverse functions and qualities than ever. For example, JEE (Java Platform Enterprise Edition)* provides JDBC (Java Data Base Connectivity), JTA (Java Transaction Architecture), JMS (Java Message Service), RMI (Remote Method Invocation) and other functions [20].

On one hand, these new middleware services ease the development, the deployment and the management of distributed systems, but on the other hand, their failures

* JCP changes J2EE to JEE just for promoting the Java brand and, in fact, does not change the architecture and mechanisms significantly. In that sense, J2EE and JEE are exchangeable in this paper.

inevitably affect the reliability of the whole system. Three main factors may lead to middleware services misbehavior, namely the middleware itself, the application code and the operator actions. For example, a network disconnection may cause failures of the communication service, of the messaging service or of the database service. For an open source middleware, its services are usually developed by different communities and their reliabilities are difficult to control because of its open source nature. Poor design decisions or implementation mistakes may also cause service failures, e.g., a database service with a reusable connection pool may crash or fall into a deadlock if the application code requests database connections from time to time but does not release them after usage. Incorrect configurations of middleware, e.g., activating too many concurrent threads, creating too many database connections, allocating memory much faster than the speed of garbage collection, may lead to incorrect middleware services or even to system crash. Generally speaking, these negative impacts on system reliability can be handled at different development stages:

- During the development, different design choices or different coding strategies result in different dependency level between applications and middleware services. Lighter the dependency is, smaller will be the impact of the middleware services failures. For example, a client can invoke the naming service to get the address of a server before each request, or before a series of requests to the server. Obviously, the failure of the naming service is more tolerable in the second case. Application developers can minimize the dependencies between the code of the application and middleware services with careful design and implementation.
- During the deployment, the delivered application has to be installed on (or bound with) a given middleware product, such as IBM WebSphere, BEA WebLogic, JBoss or JonAS. Since different products may vary in reliability, cost and risk, the selection and the configuration of the middleware product influence the reliability of the whole system. System operators must reach a trade-off between the reliability and the other concerns, like performance, cost and risk and must only pay for the fault-tolerance solutions, which are necessary in the target system.
- During the execution, different fault-tolerance mechanisms may vary in efficiency and cost [4]. If a middleware service has little or no impact on the runtime system, it is unnecessary to activate the fault-tolerance mechanisms for this particular service. Middleware vendors can therefore concentrate the resources on the most important services in order to improve their reliability, e.g., replicating the service instances, allocating more memory, or even providing alternative implementations of these services.

Obviously, all of the above reliability assurance techniques can be used if and only if the middleware services, which have a significant impact on the reliability of the whole system, can be properly identified. However, the needed “impact evaluation” is complex, time-consuming and error-prone for almost all stakeholders in the above stages, from the

application developers, the system operators to the middleware vendors. In this paper, we propose a simulation-based approach to automatically evaluating the impact that middleware services failures have on the overall system reliability. Firstly, we empirically evaluate the impact using an architectural model of middleware-based systems. Secondly, we analyze the technical challenges of service impact evaluation and then we propose a simulation-based approach with a set of rationales. Thirdly, we provide a supporting framework for automatically evaluating the middleware service impacts. It can be seamlessly integrated with multiple middleware products, including JBoss, JonAS and PKUAS, by using dynamic aspect oriented programming mechanisms so that the code of middleware products is modified implicitly and temporarily. Finally, we experiment on two JEE blueprint applications, namely JPS and ECperf.

The rest of this paper is organized as follows: Section 2 gives an empirical investigation of the impact that middleware services may have on system reliability. Section 3 explains the rationales of our approach. Section 4 discusses the implementation of the supporting framework. Section 5 presents the experiment results and analysis performed on the JPS and ECperf applications. Section 6 discusses the lessons learnt and the limitations of our framework. Section 7 introduces some related work. Section 8 concludes the whole paper and identifies the future work.

2. Illustrative Case

2.1. *Reliability and Dependency Map*

System reliability is defined as “continuity of correct service” [3] and as the likelihood that the system will remain operational (potentially despite failures) for the duration of a mission [2]. In order to analyze the service impact on system reliability, we need to choose one measurement of service reliability. So far, there is no a consensus on service reliability metrics. In this paper, the reliability of a given middleware service is calculated as the ratio of successful invocations. For example, if the reliability of a service is 80%, it means that 20 invocations out of 100 failed due to faults injected by our approach. In a middleware-based system, the service failure cannot be known by end-users directly: a failure first affects internal components and can then be propagated until the end-user. Empirically, the degree to which the failures of a service affect the overall system reliability is related to the number of components that depend on this particular service. Therefore we can use a dependency map to analyze the reliability impacts of service failures.

There are many possible types of dependency between components including direct or indirect, explicit or implicit, function-based or data-based. Since exception-handling mechanisms are popular in today’s middleware-based systems, more and more failures can be manifested by exceptions (more details will be discussed later in the section of fault model). Then, in this paper, we only consider explicit function-based dependencies, that is, if and only if component A invokes component B directly, A depends on B.

2.2. *JEE and Its Services*

JEE defines a standard application programming model for developing multi-tier, thin-client application and a standard platform for hosting applications. JEE reduces the cost and complexity of developing these multi-tier services, resulting in services that can be rapidly deployed and easily enhanced as the enterprise responds to competitive pressures. There are various standard services defined in JEE [20], as shown in Figure 1.

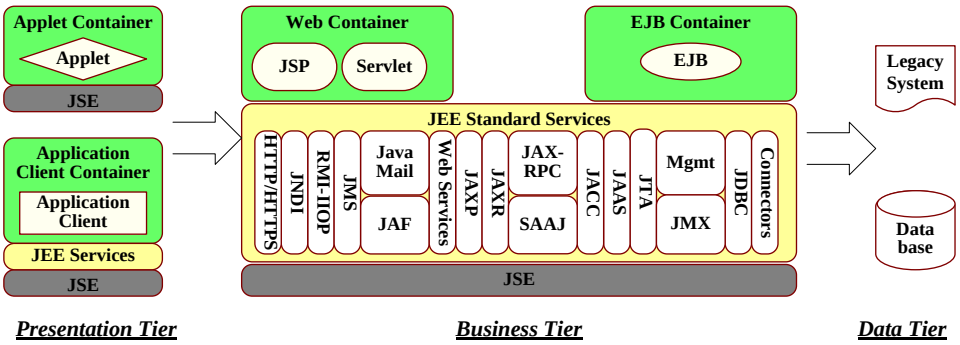

Figure 1. Services in JEE Architecture

It is worth to note that new standard services may be added in the new versions of JEE specification from time to time. For example, JAAS, JAXP (Java API for XML Processing) and Connectors were added in JEE version 1.3, and the support of web services was added in JEE version 1.4. Since the number of JEE standard services will increase continuously and many JEE application servers will support customer-defined services, the impact that middleware services have on the system performance will keep growing and the problem addressed by this paper will become more and more critical.

2.3. *Empirical Study on ECperf and JPS*

ECperf is an Enterprise JavaBeans (EJB) benchmark, initially designed to measure the scalability and performance of JEE servers and containers [22]. ECperf uses manufacturing, supply chain management and order/inventory as the “storyline” of the business problem, which is a complex and industrial strength distributed problem.

Figure 2 shows the dependency map of manufacturing domain of ECperf. It can be clearly seen that all EJBs depend on the transaction service so that these EJBs will be affected if the transaction service fails. At the same time, only three EJBs, i.e., LargeOrderEnt, WorkOderEnt and ComponentEnt, directly depend on the database service. As a result, only three EJBs will be affected immediately if the database service fails. Just considering the immediate or direct impact, it seems that the transaction service

failure is rather possible to decrease more system reliability than the database service failure.

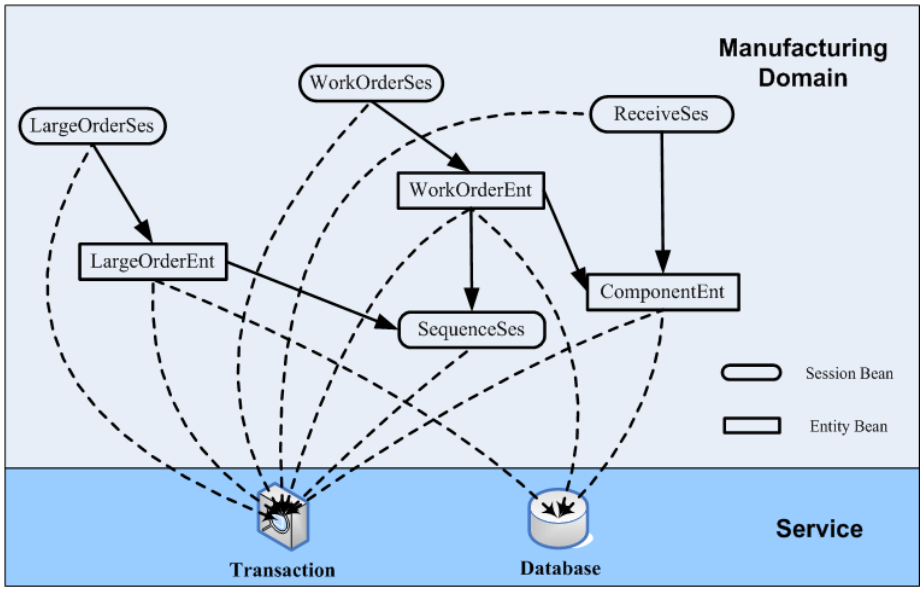

Figure 2. Reliability Impact Analysis of ECperf (partial)

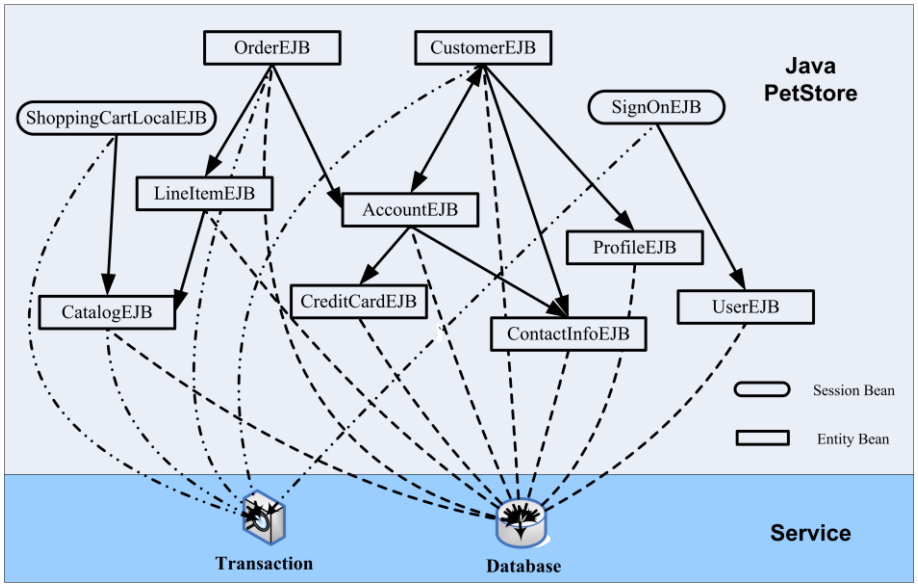

Figure 3. Reliability Impact Analysis of JPS (partial)

The Java Pet Store (JPS) is a blueprint application that demonstrates how to use the capabilities of the JEE platform to develop flexible, scalable, cross-platform e-business applications [21]. When we see the dependency map of JPS shown in Figure 3, we can find that the dependencies between EJBs and services are different from those of ECperf: more EJBs depend on the database service. This directly implies that the database service failure further decreases the system reliability. Although the naming service that is also used in JPS and ECperf is not shown in the figures, we can still conclude that the degree to which the failure of a given middleware service affects the system reliability differs from one application to another and that it is also related to the specific service that fails.

3. Approach Overview

The empirical evaluation of the impact a service failure has on the system reliability is valuable and feasible but complex, time-consuming, error-prone and imprecise because it requires an exhaustive collaboration among all stakeholders. Application developers are responsible for drawing the structure and the behavior of the application. System operators determine the candidate middleware products, the target underlying environments and the operating strategies. Middleware vendors provide the details of the middleware implementation and runtime. All of them have to work together for elaborating the dependency map. Assuming that they could effectively identify the critical middleware service and that they change the application code, or select a better middleware product or even modify the middleware implementation, they would have to further collaborate on the adjustment of the evaluation, in order to take this change into account.

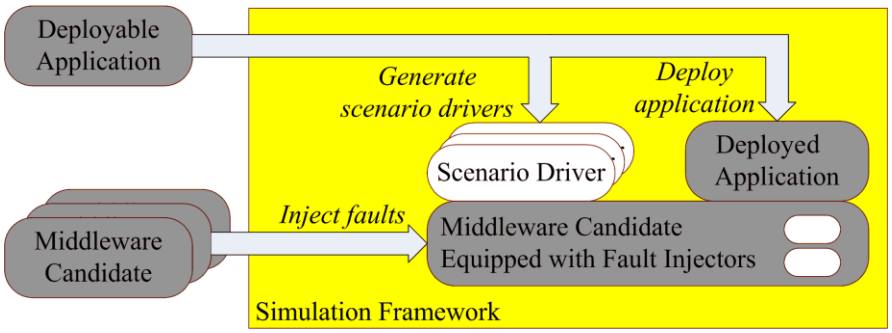

Figure 4. Simulation based Evaluation Framework

As a result, the automated evaluation of the middleware service failure impact is needed and should be done using simulation techniques when deploying the target system due to the natures of middleware based systems and reliability. In middleware-based systems, the source code of the application is usually not available to middleware vendors and system operators whereas the source code of the middleware is usually not available to application developers and system operators. The evaluation based on source code is

therefore impracticable. More important, the reliability is a runtime property and then the evaluation should be based on runtime data. So, we propose a simulation-based evaluation framework as shown in , which can be used in four steps:

[1] Test client configuration: Application developers provide the deployable application and collaborate with system operators to determine the use cases to be evaluated. For simulating the use cases, a set of scenario drivers have to be developed. Usually, the test cases simulating the important use cases have already been developed when testing the application and can therefore be reused as scenario drivers with little or no revision.

[2] Service failure configuration: System operators select a product from a set of middleware candidates and collaborate with middleware vendors to determine what kind of fault to inject, and into which middleware services the fault should be injected.

[3] Simulation: Firstly, the middleware product is started and injected with a given kind of faults for simulating service failures at runtime according to the service failure configuration. Secondly, the application is started on the given middleware product and the scenario drivers will be started at the given time according to the test client configuration. Then the faults will be injected, the service will fail and the use case may fail too. All runtime data will be collected.

[4] Result analysis: After acquiring reliability evaluation results, system operators, application developers and middleware vendors can study the result carefully to find out the best-of-the-breed fault tolerant configuration, select another middleware product or redesign some parts of the system.

The second and the third steps are iterative. Only one service is injected with one or more faults in one iteration because our approach aims at the impact of a single service. The number of services to be analyzed determines the times of iteration. .

3.1. *Fault Model*

The faults in middleware based systems can be divided into three levels, including underlying environment, middleware and application. The underlying faults may come from operating systems, databases, networks and programming language runtimes, such as the physical errors of processor and memory (e.g., bit flip caused by radiation), no hard disk space, network traffic jam and so on. The middleware faults come from middleware services, such as the defect of service implementation, overflow errors in hash maps, incorrect synchronization, etc. The application faults come from the implementation of the application, i.e., the defects or bugs.

For capturing the faults coming from three levels, we use the concept of “exception” provided by modern programming languages. In Java, for example, “when a program violates the semantic constraints of the Java programming language, the Java virtual machine signals this error to the program as an exception” [24]. Some experiments in [4] show that almost all underlying faults can be manifested by Java exceptions. At the same

time, in terms of the philosophy of exception handling, throwing and catching exceptions become the most popular mechanisms to deal with faults when programming middleware products and applications. Consequently, most of the faults coming from the underlying, middleware and application levels can be captured by exceptions. After analyzing the bug repository of some open source JEE application servers, we find that more than 70% middleware failures are manifested by exceptions [29]. Moreover, there are two advantages of manifesting faults as exceptions on evaluating service impacts on reliability of distributed system: one is both software faults and operator mistakes are the main causes of failures in distributed systems [17]; another is exceptions can be simulated and controlled in a fine-grained, easy and precise manner.

3.2. *Simulation of Service Failure*

Fault injection is a technique to observe a system's behavior when a special kind of input, i.e., faults, is introduced into the system [1]. Being fast (accelerating the occurrence of faults, errors and failures), cost-effective (without extra hardware) and well controlled (when, where and what to inject), software implemented fault injection is employed for simulating service failures in our approach.

Technically, some pre-treatment and post-treatment codes will be added at the beginning and the end of the implementation of service methods, respectively. These codes should have the ability to simulate every possible service failures. Avizienis et al. [3] classify failures into content failures, timing failures, halt failures and erratic failures, and we can simulate each of them. Content failures can be simulated either by throwing exceptions in the pre-treatment code, or by modifying result values in the post-treatment code. Timing failures can be simulated by delaying the return of methods in the post-treatment code. Halt failures can be simulated by adding non-ending loop in either pre-treatment code or post-treatment code. Erratic failures can be simulated by firstly delaying some time and then throwing exceptions or modifying result values in the post-treatment code.

There are two failure injection modes: durative mode and random mode. In the durative mode, the service failures are injected continuously during some specified periods of time. In the random mode, the test time is divided into time unit and the service failures are injected randomly at each time unit. The ratio of the total injection time, i.e. the service reliability, is determined by the configuration. As a result, changing the configuration can predict the effectiveness of the reliability improvement of a given service. In some cases, loosing service dependency leads to reduce the frequency and number of service invocations. It means that changing the injection ratio can also simulate the reliability improvement of application codes.

3.3. *Measurement of Service Impact on Reliability*

Use cases are means to specify required usages of a system [18]. They describe the interactions between end-users and computer systems. The combination of the use cases

with the corresponding internal execution traces of the target system enables the detailed evaluation of the impact of a given service. For example, in JPS, a test client named TestListCategories corresponds to the use case named ListCategories. It includes getting the item list and their general information. This use case represents the scenario where a customer opens the website and browses the product catalog. Then if this use case works well, every customer can browse the catalog. Observing the output of TestListCategories during the simulation of service failures helps us find out whether customers can still browse the product catalog, even if some services fail.

By testing each use case and grouping the measurement results, we can easily find out which use cases are affected and to what extent they are affected by the middleware service' failures. The service impact on reliability of the whole system can be simply calculated as:

$$IS_i = \sum_{j=1}^{|U|} (IU_{i,j} \bullet PU_j \bullet WU_j)$$

Where IS_i is the impact of failures of service i on the whole system, $IU_{i,j}$ is the impact of failures of service i on use case j, PU_j is the probability of using use case j, and WU_j is the weight of the use case j (the sum of all use cases should be 1). $IU_{i,j}$ can be obtained by calculating the failure proportion of use case j when service i is failed. PU_j is the objective factor of use case j which reflects its frequency of occurrence in the real scenario while WU_j is the subjective factor of use case j which reflects the importance to some stakeholders. For example, the use case "Order" is much more important than the use case "List Categories" in JPS application, therefore, WU of the "Order" is larger than WU of the "List Categories". $IU_{i,j}$, PU_j, and WU_j range from 0 to 1.

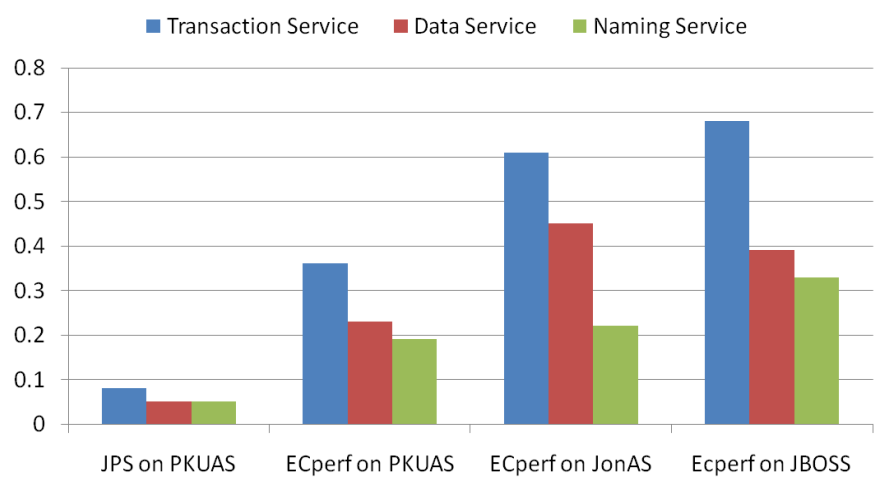

Figure 5. Service Impacts on System Reliability

Using the above formula, Figure 5 shows the service impacts on system reliability in all experiments (assuming that all use cases have the same weight and probability). We can conclude that the application's implementation is crucial to the degree of the reliability decreasing in the presence of service failures. Furthermore, the implementation's diversity and variety determine the reliability decreasing degree will be different among different parts of an application and different applications in the presence of different service failures.

4. Implementation

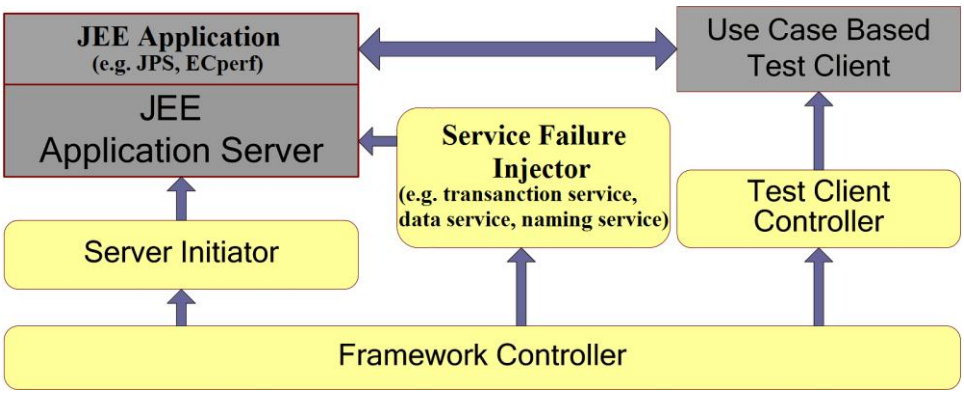

Figure 6. JEE Implementation of the Evaluation Framework

shows JEE implementation of the simulation-based evaluation framework. The JEE Application Server is the middleware to be evaluated (currently PKUAS, JBoss and JonAS are integrated into the framework). The JEE Application is the target application, e.g., ECperf and JPS in our experiments, and the use case based test clients are programs invoking server-side methods to satisfy specific requirements. Server Initiator is responsible for adding failure injection code into services implementation of JEE Application Server when services are loaded, and for controlling the start and stop of the application server. Service Failure Injector triggers the injection of service failures by invoking special methods inserted into the service implementation by Server Initiator. Test Client Controller is responsible for loading test client programs, generating test workload, observing the application's output and calculating the system reliability. Framework Controller coordinates Server Initiator, Service Failure Injector and Test Client Controller as shown in .

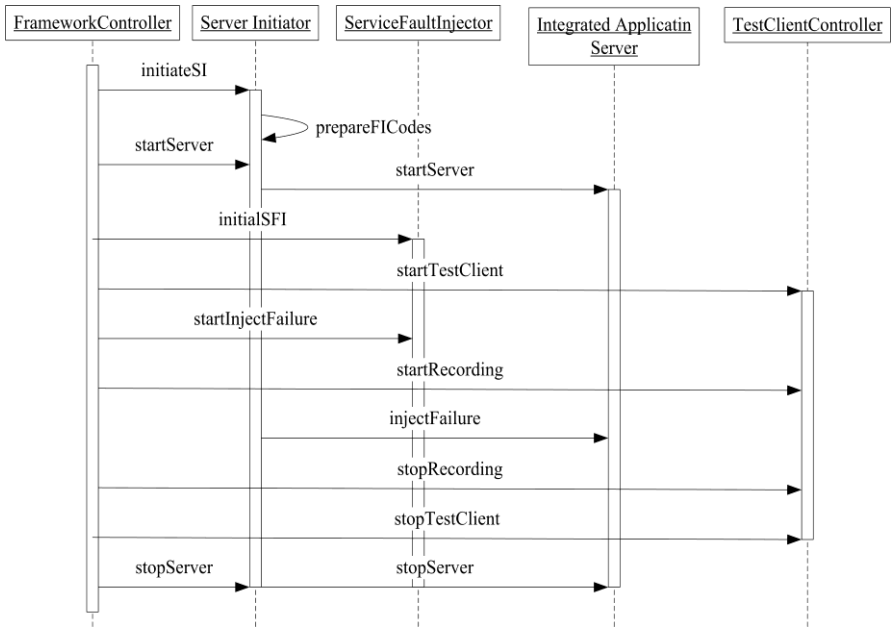

Figure 7. Control Flow in the Evaluation Framework

Since the framework has to evaluate multiple JEE application servers for a given application, the most important challenges of JEE implementation are: i) how to add fault injection code into the service implementation, ii) how to modify the class loading mechanism in order to ensure the use of the modified service implementations, and iii) how to configure the Server Initiator so that it can start and stop the application server in a general enough manner.

The modifications of the original service implementation, i.e., adding pre-treatment and post-treatment codes, may involve a large number of source files and the scattering of fault injection codes causes a great damage to the manageability of services implementation. These issues are just the motivation of Aspect Oriented Programming (AOP) [11], which advocates the use of aspects to implement crosscutting concerns whose codes scatter over the whole system originally. Moreover, in practice, the source code of middleware services is usually not available to all stakeholders except to the middleware vendors. Weaving the pre-treatment and post-treatment codes as aspects into the original implementation of middleware services dynamically can avoid some security and license issues.

Consequently, we use the Javassist (JAVA programming ASSISTant) toolkit [7] in Server Initiator to modify the implementation classes of the services during the class loading. Javassist makes the Java byte code manipulation simple. It is a class library for editing byte codes in Java; it enables Java programs to define a new class at runtime and to modify a class file when the JVM loads it. illustrates such modifications of

middleware service implementations and depicts the addition of a variable into the class SmartCtx and of a code fragment into the beginning of the "lookup" method. The fault injection is configurable: the application server, the service and its implementation method, the begin and stop times of the injection, as well as the injection mode (durative or random) can be specified by a GUI tool. Figure 9 shows that only the "notify" method of TraceEjb (it is not an EJB but a small private service) in JonAS will be injected with faults. The information about the implementation of an application server is retrieved automatically by Java Reflection APIs, which can acquire almost all information of a Java class except the internal details of a method body (generally speaking, Java classes are read-only in Java Reflection while writable in Javassist).

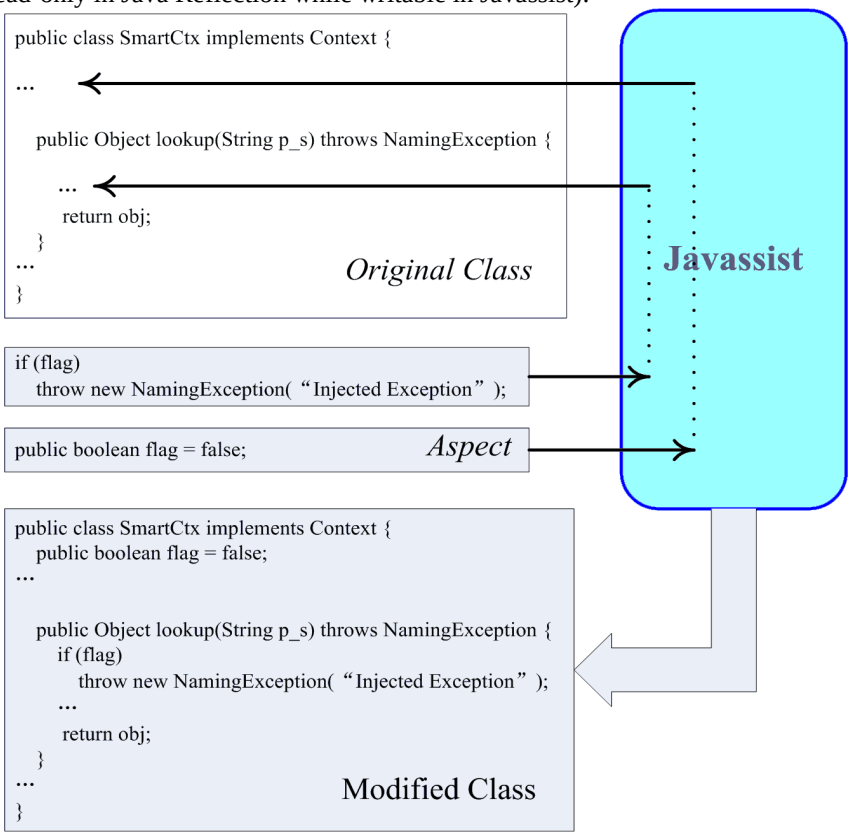

Figure 8. Code Modifications using Javassist (the source codes of service are not needed)

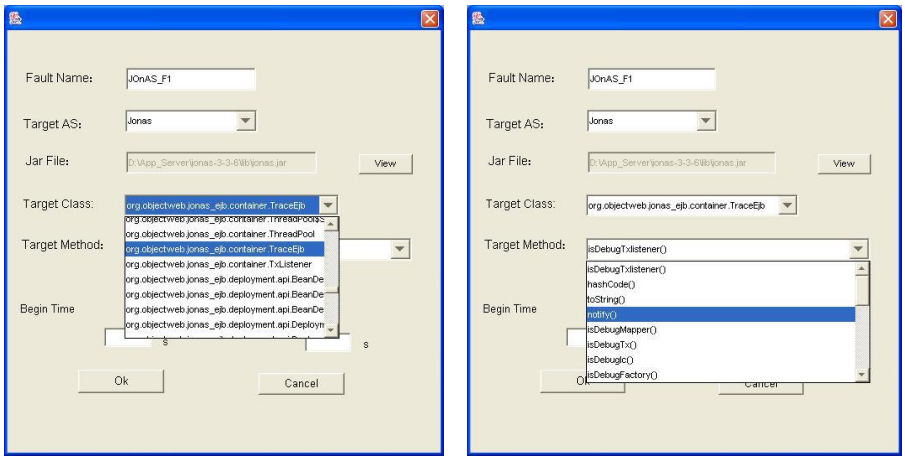

Figure 9. GUI for Configuring Fault Injection

Each application server has its own class loading mechanism and we must make sure that the modified classes are actually used at runtime. For example, JBoss employs a new class loader architecture and uses a collection of unified class loaders to load classes of deployed applications and services [8]. We implement a special class loader named FIClassLoader, which takes responsibility for loading the classes modified by Server Initiator. The class loaders of the integrated application server must be modified. When they load a class, they have to ask FIClassLoader to load the class first and load the class by themselves only when FIClassLoader cannot load that class. Because Server Initiator controls the start and stop of the integrated application server, it must know how to do so. And this is easily implemented by configuring corresponding class name and method name to start and stop the application server in a configuration file.

5. Experiment

Due to space limitations, we only present our experiments on the naming service, database service and transaction service of PKUAS, JBoss and JonAS in ECperf and JPS. The failures are injected into each service separately and the injection is specific to services. For example, javax.naming.NamingException and javax.transaction.SystemException are injected into the naming service and transaction service respectively. Assuming that the reliability of all services is the same – 80%, i.e., 20 out of 100 invocations failed due to faults injected to a given service. Both durative mode and random mode are tested but, here, we only show the random mode because it reflects the real case better according to our findings in bug repository analysis of JEE application servers [29]. In Figure 10-14, the x-axis identifies the use cases of JPS or ECperf and the y-axis is the failed request ratio when injecting faults into the given service. When analyzing the experiment results, we will present some studies on the

source codes and deployment descriptors of ECperf, JPS, PKUAS, JBoss and JonAS for better comprehension.

(F1) Static reliability impact analysis may produce wrong results.

It is very clear in all figures in this section that the failures of different services have different reliability impacts on the same use case. Common sense explains that as discussed previously. It is noteworthy that the transaction service has stronger impacts than the other two services in almost all use cases, which contradicts the static analysis described in Section 2.3. In fact, the transaction service is used to guarantee the proper termination of a sequence of operations and any failure of the transaction service during this period may lead to a failure of the whole sequence of operations. By contrast, the naming service and the database service are used by isolated method invocations that last for a very short period of time. In other words, a failure of the transaction service affects a sequence of multiple invocations whereas a failure of the naming service or of the database service only affects a single invocation. Failures of the transaction service have consequently a more significant impact on overall system reliability. Although it is difficult to identify such a primary reason by both static analysis and dynamic analysis, the dynamic analysis can at least reveal the failure caused by that primary reason while the static analysis cannot.

(F2) The time a service failure happens has a significant impact on the failure of the use case depending on the service.

It is also very clear in the experiment results that the failures of one service have different reliability impacts on different use cases. Considering the transaction service as an example, most use cases are affected by its failures, except “Cart Add”, “Cart List” and “Cart Delete”. From their source codes we can see that the “Cart List” and “Cart Delete” do not actually use the transaction service. Contrarily, the “Cart Add” uses it, but it seems that there is no impact of the transaction service. This is due to the time when a service fails.

When executing the same testing configuration at different times, we find that the same service always has different impacts on the same use case in different executions. During the “Sign On” process, the invocation of the database always uses the same customer’s username and password to send out the Sign On request. According to the JEE specification, an entity EJB instance will be initialized and synchronized with the database when it is used for the first time, but the synchronization will be done again only if some properties have changed. The EJB instances used in the “Sign On” are all loaded during the first request and the later requests do not actually perform any database operations. That is to say, if the faults of the transaction service are injected after the first request in the “Sign On” case, they do not lead to failures of the case.

Actually, this reveals the fact that the reliability cannot be precisely evaluated because failures cannot be predicted (otherwise the failures could be prevented or recovered). In that sense, our simulation-based evaluation and other approaches only

evaluate the "trend" or "possibility" of system reliability. However, we believe the work remains valuable enough. Furthermore, a more precise evaluation can be obtained by executing the same testing configuration at different times and making statistics on the results (the statistics algorithm is under development).

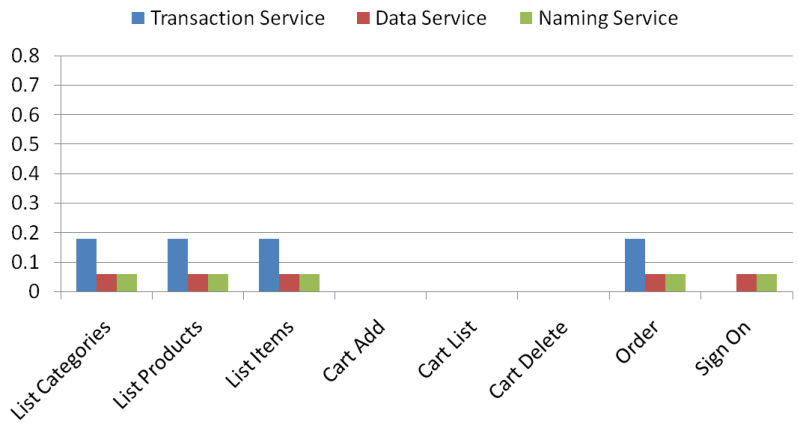

Figure 10. JPS on PKUAS (80%)

(F3) The same service provided by different middleware products, even compliant with the same standard, has different reliability impacts.

Figure 11, Figure 12 and Figure 13 show the experiment results on PKUAS, JonAS and JBoss, respectively. We can see that the same service with similar kind of faults injected into different application servers with similar ratio and time lead to different reliability impacts on the same scenario in the same application. The reason is very clear when we read the source code of the three application servers. Usually, a middleware service is not invoked by the application code directly. These invocations are mediated by JEE containers. Containers are a special and critical part of modern middleware, responsible for supporting component-based development methods (e.g., incarnating a special component model, like EJB and CCM), providing a virtual runtime environment for application components (e.g., managing lifecycle of a component), and coordinating middleware services and application components. PKUAS, JonAS and JBoss have different design and implementation of JEE containers, including the integration, coordination and management of middleware services, which lead to the different service impacts.

This finding is very important to the selection of middleware products compliant with some standards, in which a set of services is standardized for the sake of application portability. Usually, such standardization just defines the function of a service, but not its quality. That is to say, an application can indeed run on different JEE application servers, JBoss or JOnAS, but will provide different level of reliability: some failures occurring in

JBoss may not occur in JOnAS. Selecting the proper middleware product may thus avoid the risk and cost of some middleware failures and our approach helps to do such a selection.

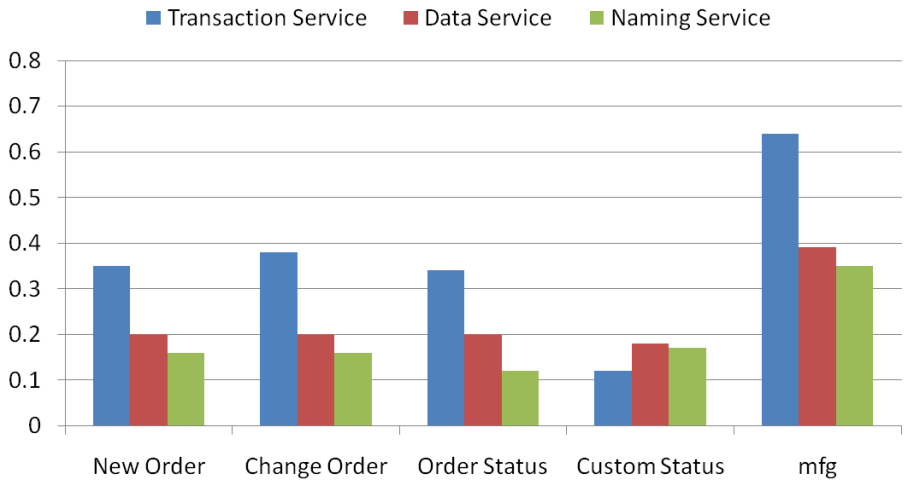

Figure 11. ECperf on PKUAS (80%)

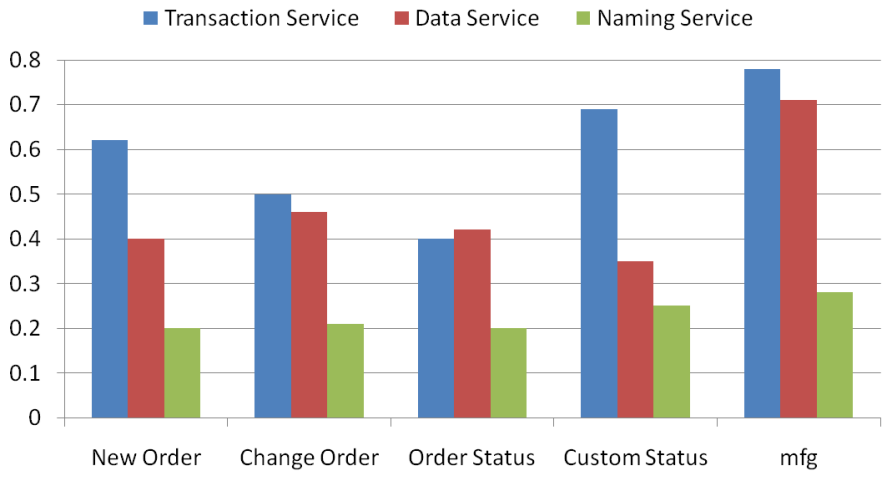

Figure 12. ECperf on JonAS (80%)

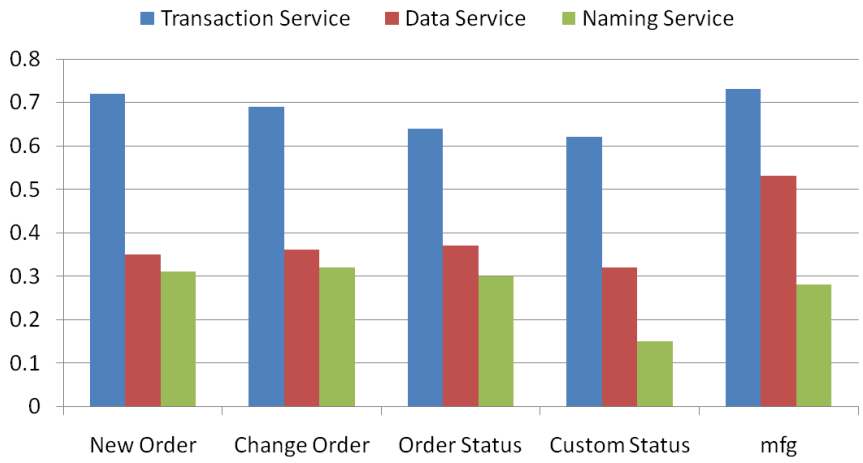

Figure 13. ECperf on JBoss (80%)

(F4) The effectiveness of reliability improvement can be predicted by changing fault injection ratio.

Through Figure 14, we can evaluate how much the system reliability can be improved if a given service reliability is improved from 70% to 80% and to 90%. For example, the impact that the transaction service has on the "New Order" is consequently decreased from 0.691 to 0.352 and 0.081 respectively. Furthermore, the impact value and its decreasing degree can reflect the strength of the dependency between a use case and a given service. For instance, the impact that the transaction service has on the "Customer Status" is decreased from 0.281 to 0.121 and 0.04 respectively, which are much smaller than those of the "New Order" while their decreasing degrees are similar. This implies that the "Customer Status" has a lighter dependency on the transaction service. This can also be obtained by analysis of the source codes of the two use cases.

This finding is very valuable for improving reliability of open source middleware. Today's open source middleware usually integrates many third-party services. Considering the liberty of open source, it is hard to ask the third-party to improve their reliability without convincible reasons. We argue that the prediction can provide such convincible data. On the other hand, replication is a popular but expensive fault-tolerant solution in today's open source middleware. Since the replication can be considered as an improvement of the reliability to some extent, the prediction helps to evaluate the return of investment of replicating a given service.

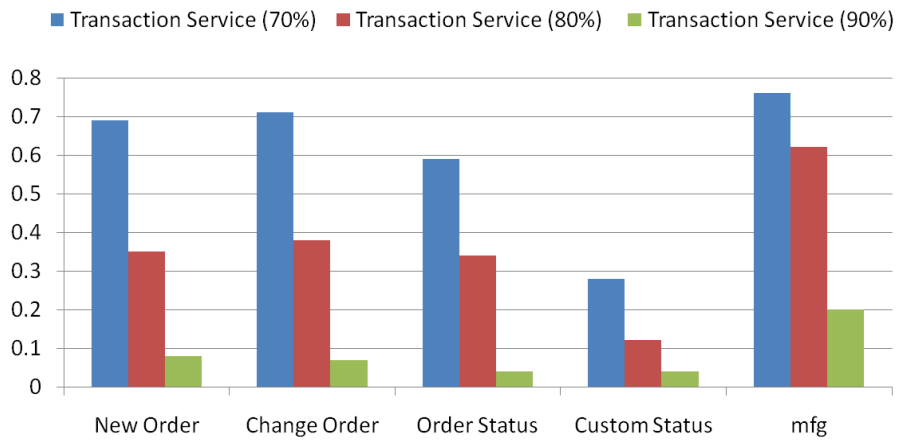

Figure 14. ECperf on PKUAS (Comparison with different service reliability)

(F5) Timing failure with different delay time has different reliability impacts.

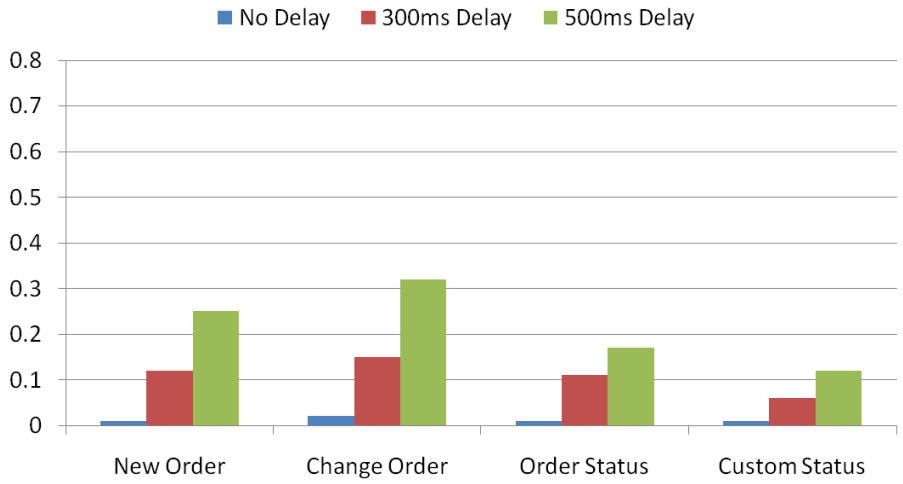

Figure 15. Timing Failure of Transaction Service on ECperf

The above experiments do not consider the timing failures, which is a usual type of failures. In fact, our approach focuses on the impact of service failure, instead of the root cause. Timing failure can be considered as a type of root cause and has the same features with the above experiments. We simulate timing failures by delaying the return of methods in the post-treatment codes of the transaction service. The delay time is set to 0 ms (milliseconds), 300 ms and 500 ms respectively. On the client side, we set the request

timeout to 1000 ms. That is to say, if the response time is greater than 1000 ms, we regard it as a failure. Figure 15 shows the results obtained on PKUAS, which reflect the common sense: longer delay time leads to bigger failure ratio and then stronger reliability impact.

6. Discussion

The interrelationship between middleware services has to be considered for better precision. For example, the failure of the database service will cause the failure of the transaction service. It means the impact of two or more services is smaller than the sum of the individual service impact. Such correlated faults are hard and sometimes impossible to evaluate precisely. Based on our experience on automated recovery of correlated faults between the database service and transaction service [13], we believe analyzing execution traces of the whole system is a promising method to find such interrelationships.

The dependency map is a fundamental tool for further analysis and decision-making regarding the reliability improvement. Now, we can automatically construct the dependency map used in the simulation, like JAGA [6]. The calculation of the impact that different service failure may have on the overall application reliability is too simple and may not accurately reflect the real situation. The dependency map helps to leverage mathematic reliability models, which enhance the reliability estimation of middleware-based systems. Now, we are trying to integrate SBRA (Scenario based Reliability Analysis) [27] with some extensions, such as introducing service components and dismissing the correlation between services.

The analysis described in the above section requires a detailed study on the source code and the deployment descriptor. There are around 400 source files in the ECperf application and even more in the JPS case. Performing such analysis remains quite error-prone and time-consuming. Moreover, this kind of empirical analysis cannot result in a precise reliability impact measure. On the other hand, our simulation-based evaluation framework can give more precise and visual results, while avoiding the tedious empirical evaluation work.

7. Related Work

Some researches identify the importance of analyzing middleware services' behavior in the presence of low-level failure and the reliability of middleware itself, such as the failure mode analysis of CORBA service implementations [14] and the measuring and modeling availability of application server [25]. However, our work focuses on the higher level: observing the application's behavior under the condition that middleware services fail.

There is much research work on reliability analysis and estimation. Markov Models are widely used to capture the system states and its transitions [27]. The main limitation of Markov Models is that it is hard to identify the large number of system states of

complex systems. Some work focuses on assessing the reliability of component-based applications, such as Program Dependency Graph and Fault Propagation Analysis [26], which generates dependency graphs from source codes. Similarly, the Component-Based Reliability Estimation [12] approach requires test information and test cases and uses path-based reliability calculation. An alternative approach is the Scenario-Based Reliability Analysis [28], which constructs a Component-Dependency Graph and provides the corresponding algorithms to analyze the application reliability. However, these efforts cannot be directly applied to middleware-based systems because they do not take the middleware's impact on system reliability into account.

Fault injection is also widely used to evaluate the reliability of computer systems [9]. Researchers and engineers have created many novel methods to inject faults, which can be implemented in both hardware and software. As to the software fault injection, Jaca [15] is a fault injection pattern system based on Java reflection and Javasisst. It injects faults by modifying either method parameters or return values. The main difference between Jaca and our framework is that Jaca does not have an explicit fault model and the consequential behavior of fault injection is undetermined, i.e., the event reflecting that a fault is injected does not imply that a fault will definitely occur at the service boundary. So Jaca cannot be used to simulate middleware service faults. On the contrary, our framework injects exceptions directly into the service interface and the exceptions will be caught by the applications, which fit the natures of middleware-based systems. We tried to integrate Jaca into our framework but we finally had to implement a new fault injection mechanism, which learns a lot from Jaca. APFI [5] and JAGA [6] also inject failures into middleware for discovering fault paths and constructing recovery maps at the application level, while we use fault injection to evaluate service impacts on system reliability and focus on the middleware level. However, we will introduce the similar capabilities of APFI and JAGA into our framework for further analysis and reliability improvement.

8. Conclusion and Future Work

The proliferation of middleware technologies makes middleware one of the most common infrastructures of distributed systems. Technically, middleware could be viewed as a collection of common services, and thus the reliability of middleware services becomes a key factor influencing the reliability of the whole system. If application developers, system operators and middleware vendors try to guarantee or improve system reliability, they have to identify the services that have a significant impact on the overall system reliability. In this paper, we present a simulation-based framework to automatically evaluate such impacts. Our solution basically captures faults as exceptions, simulates service failures by software implemented fault injection, generates workload using test clients and gathers runtime data in order to calculate the reliability impacts of

each service failure. The framework is implemented in JEE and can integrate multiple JEE application servers, such as PKUAS, JBoss and JonAS.

We believe many limitations previously discussed can be addressed if we refine our approach on the basis of SM@RT [10], which can dynamically construct a software architecture model of a runtime system and change the runtime system if the model is changed. First of all, SM@RT gives the evaluation framework fine-grained control over the target system so that faults can be injected into more locations and more runtime data can be collected. Recall the calculation of service impact in Section 3.3, more factors can be considered and usually different factors need different runtime data. For example, the critical level of a fault reflects the severity of consequence when the fault occurs. If we try to take this factor into the calculation, we have to collect more runtime data to evaluate whether some system qualities, like the response time and throughput, decrease and to what extent when a fault is injected. Secondly, SM@RT enables the evaluation framework compliant with MOF (Meta Object Facility), which is the most popular modeling standard in practice, so that many model-based reliability analysis methods can be integrated, like SBRA. Thirdly, SM@RT brings the evaluation framework some potential applications, e.g. evaluating whether the impact of middleware service failure decreases or increases before re-configuring the middleware and application.

References

[1] Arlat, J.; Aguera, M.; Amat, L.; Crouzet, Y.; Fabre, J.-C.; Laprie, J.-C.; Martins, E.; Powell, D. Fault Injection for Dependability Validation: A Methodology and Some Applications. IEEE Transaction on Software Engineering, 16(2), 1990, pp.166-182

[2] Arun K. Somani, Nitin H. Vaidya, Understanding Fault Tolerance and Reliability, Computer, IEEE, Vol. 30, Issue 4, April 1997, pp 45-50.

[3] Avizienis, A., J.C. Laprie, B. Randell and C. Landwehr (2004). 'Basic Concepts and Taxonomy of Dependable and Secure Computing', IEEE Transactions on Dependable and Secure Computing, Vol.1, No.1, pp 11-33.

[4] Candea, G., James Cutler, Armando Fox, Improving Availability with Recursive Microreboots: A Soft-State System Case Study, Performance Evaluation Journal, vol. 56, nos. 1-3, March 2004.

[5] Candea, G., Mauricio Delgado, Michael Chen, Armando Fox, Automatic Failure-Path Inference: A Generic Introspection Technique for Internet Applications, the 3rd IEEE Workshop on Internet Applications (WIAPP), San Jose, CA, June 2003.

[6] Candea, G., E. Kiciman, S. Zhang, P. Keyani, A. Fox. JAGR: An Autonomous Self-Recovering Application Server. In Proceedings of Autonomic Computing Workshop and Fifth Annual International Workshop on Active Middleware Services (AMS'03), June 2003.

[7] Chiba, S. Javassist: A Reflection-based Programming Wizard for Java, in Proc. of the Workshop on Reflective Programming in C++ and Java, Canada, 1998.

[8] Fleury, M. and F. Reverbel, The JBoss Extensible Server, in Proceedings of Middleware 2003, LNCS 2672, pp. 344–373, 2003.

[9] Hsueh, M., Timothy K. Tsai, and Ravishankar K. Iyer, Fault Injection Techniques

and Tools, IEEE Computer, April 1997.

[10] Gang Huang, Hui Song, Hong Mei. SM@RT: Applying Architecture-based Runtime Management of Internetware Systems. International Journal of Software and Informatics, 2009, 3(4):439~464.

[11] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, Aspect Oriented Programming, 11th European Conference on Object-Oriented Programming (ECOOP'97), Finland, June 1997, pp. 220–243.

[12] Krishnamurthy, S., A.P. Mathur, On the Estimation of Reliability of a Software System Using Reliabilities of Its Components, in Proc. Of 8th International Symp. Software Reliability Engineering (ISSRE), 1999, pp.146-155.

[13] Liu, T., Gang Huang, Gang Fan, Hong Mei, The Coordinated Recovery of Data Service and Transaction Service in J2EE, In Proceedings of 29th Annual International Computer Software and Applications Conference (COMPSAC), Edinburgh, Scotland, July 2005, pp. 485-490.

[14] Marsden, E. and J. Fabre, Failure Mode Analysis of CORBA Service Implementations, Middleware 2001, LNCS 2218, pp. 216–231, 2001.

[15] Martins, E., Cecilia M.F. Rubira, Nelson G.M.Leme, Jaca: A reflective fault injection tool based on patterns, in Proceedings of the International Conference on Dependable Systems and Networks (DSN'02), 23-26 June 2002, pp. 483- 487.

[16] Mei, H. and G. Huang. PKUAS: An Architecture-based Reflective Component Operating Platform, invited paper, 10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS), 2004, Suzhou, China.

[17] Patterson, D A, A Brown, P Broadwell, et al. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies, UC Berkeley Computer Science Technical Report UCB//CSD-02-1175, 2002.

[18] Object Management Group, UML 2.0 Superstructure Specification, v 2.0, 2004.

[19] SUN Microsystems, Enterprise JavaBeans Specification, Version 2.0, 2001.

[20] SUN Microsystems, Java Platform Enterprise Edition Specification, Version 5, http://java.sun.com/javaee/technologies/javaee5.jsp.

[21] SUN Microsystems, Java Pet Store, http://java.sun.com/reference/blueprints.

[22] SUN Microsystems, ECperf Specification, Version 1.1, Final Release, http://java.sun.com/j2ee/ecperf/.

[23] Sun Microsystems, Java 2 Platform Enterprise Edition Management Specification, 2002.

[24] Sun Microsystems, The Java Language Specification, 3rd edition, ADDISON-WESLEY, 2005.

[25] Tang, D., D. Kumar, S. Duvur, O. Torbjornsen, Availability Measurement and Modeling for An Application Server, International Conference on Dependable Systems and Networks (DSN), 2004, pp. 669- 678.

[26] Voas, J. Error Propagation Analysis for COTS System, IEEE Comput. Control Eng. J., Vol. 8, No.6, pp.269-272, Dec. 1997

[27] Whittaker, A. and M. Thomason, A Markov Chain Model for Statistical Software Testing, IEEE Transactions on Software Engineering, Vol. 20, No. 10, pp. 812-824, Oct, 1994.

[28] Yacoub, S., Bojan Cukic, Hany H. Ammar, A Scenario-Based Reliability Analysis Approach for Component-Based Software, IEEE Transactions on Reliability, Vol.53, No.4, Dec. 2004, pp.465-480.

[29] Junguo Li, Gang Huang, Jian Zou and Hong Mei.Failure Analysis of Open Source J2EE Application Servers. International Conference on Quality Software (QSIC), USA, 2007, pp 198-208.

Gang Huang received his PhD degree in Computer Science from Peking University in 2003. He is currently an associated professor at the School of Electronics Engineering and Computer Science, Peking University, Beijing, P.R. China. His current research interest includes distributed computing with focus on Web Services, J2EE and CORBA, component based software development with focus on software architecture and component framework. He is a member of IEEE. Email: huanggang@sei.pku.edu.cn.

Weihu Wang is a Master student at the School of Electronics Engineering and Computer Science, Peking University, Beijing, P.R. China. His current research interest focuses on runtime software architecture. Email: wangwh08@sei.pku.edu.cn

Tiancheng Liu is a Ph.D student at the School of Electronics Engineering and Computer Science, Peking University, Beijing, P.R. China. His current research interest focuses on reliability of middleware based systems. Email: liutch@sei.pku.edu.cn

Hong Mei received his PhD in Computer Science from Shanghai Jiaotong University in 1992. He is currently a full professor at the School of Electronics Engineering and Computer Science, Peking University, Beijing, P.R. China. His current research interest includes software engineering and software engineering environment, software reuse and software component technology, distributed object technology, software production technology, and programming language. He is a senior member of IEEE. Email: meih@pku.edu.cn