Coordinated Recovery of Middleware Services: A Framework and Experiments*

Hong Mei, Gang Huang, Tiancheng Liu, Junguo Li

(Key Laboratory of High Confidence Software Technologies, Ministry of Education, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

Abstract Being the most popular runtime infrastructure for distributed systems, middleware can be viewed as a collection of common services. Since the development, deployment and maintenance of distributed systems rely largely on middleware services, the failure of middleware services puts a significant impact on the reliability and availability of the whole system. Though recovery-based fault tolerance is an effective way to improve the reliability of middleware services, it is far away from practice mainly because of the high complexity and cost of the recovery of correlated failures between interdependent services. In this paper, a framework for detecting and recovering the correlated failures of middleware services in an automated way is presented. First, the problem is investigated from two perspectives, i.e., analyzing the role and impact of middleware services and illustrating a set of correlated failures in J2EE standard services as motivating examples. Then, a general coordinated recovery model is constructed with the elements necessary and sufficient for detecting and recovering correlated failures in middleware services. The supporting framework is demonstrated on three J2EE application servers, i.e., PKUAS, JBoss and JOnAS, one by one without fundamental modifications. Finally, based on the three enhanced application servers, many cases on J2EE common services, including the transaction service, database service, naming and directory service, security service and messaging service, are studied. The experiment results show the effectiveness and applicability of the framework presented in this paper.

Key words: reliability; recovery; middleware; J2EE


1 Introduction

The rapid evolution and pervasiveness of network and distributed applications make middleware technologies proliferate. Middleware can be viewed as a collection of common services which provide both applications and application developers with
good support in common functions such as communication, transaction and security. Since the development, deployment and maintenance of distributed systems rely largely on middleware services, the dependability of middleware services becomes one of the key factors influencing the reliability and availability of the whole system. For example, in [36], we analyze the bug repositories of three popular open source J2EE application servers, JBoss, JOnAS and Geronimo, and find that failures caused by services have always existed and account for more than half of all failures.

Being a promising approach, recovery-based fault tolerance can improve system dependability by fast recovery of failures. But the recovery of middleware services mainly focuses on individual recovery currently: once a service fails, it is recovered alone. The interdependent relationships among services are not taken into account. As we know, there exist interdependent relationships among middleware services. For example, the transaction service depends on the data service, which results in the consequent failures of the transaction service after the data service fails. In such interdependent services, once a service (called depended-on service) fails, the services (called depending services) depending on it, will fail also. This set of failures is called correlated failures and the corresponding faults in services are called mutually dependent faults [22] or dependent faults. Although individual recovery can make the service work again in most cases, it is still necessary to coordinate the recovery process, which may involve several interdependent services, to improve recovery efficiency. Firstly, the operations affected by these failures cannot be recovered without the collaboration between depended-on and depending services. So the applications using these services would be unavailable from the time the failures occur to the time when all services are recovered. Secondly, some interdependent relationships are set and initialized when the services start, and when one service fails, others should be re-initialized so that the invalid references and other environmental variables can be updated. Thirdly and lastly, as the startup of interdependent services is sequential [31], the recovery of interdependent services is also sequential and should be coordinated and guaranteed by a coordinator. So the recovery on middleware services should be performed in a coordinated manner.

In this paper, we propose a framework to coordinate the recovery of middleware services, including fault detectors, recoverable services, recovery policies and a central coordinator. The framework is demonstrated on J2EE application servers, which are popular middleware today (we consider middleware and application server interchangeable in this paper). The framework is general enough for J2EE since it is implemented in PKUAS [19], our own J2EE application server, at first and then migrated to JBoss [14] and JOnAS [15], the top 2 open source J2EE application servers, without any fundamental modifications. Based on these three enhanced application servers, we study a set of cases of the mutually dependent faults and coordinated recovery in J2EE common services, including the transaction service and data service, the naming service and security service, and the messaging service and transaction service.

The rest of this paper is organized as follows. Section 2 introduces the background of our work, including the role and impact of middleware services, and the Recovery Oriented Computing. Section 3 discusses the analysis of mutually dependent faults and correlated failures in middleware services. Section 4 presents the framework with
Hong Mei, et al.: Coordinated recovery of middleware services: A framework ... 103

a general coordinated recovery model and a set of key elements. Section 5 describes the experiments in J2EE common services and Section 6 provides some related work and discusses the contributions and limitations of our work. Section 7 concludes the whole paper and identifies the future work.

2 Background

2.1 Reliability and availability of Middleware Based Systems

Nowadays, more and more complex software systems are built on middleware, which is a software layer between the applications and underlying operating systems, as shown in Figure 1[18]. For reducing the complexity and cost of the development and evolution of distributed systems, middleware encapsulates plentiful functions common to distributed systems, which can be divided into three categories. Firstly, middleware encapsulates plentiful capabilities to manage underlying computing resources while these functions are traditionally considered as the major functions of distributed operating systems. Secondly, though middleware originated from problems common to most of distributed systems, it implements much more functions only usable in a specific application domain, such as telecommunications, finances, retails, etc. Thirdly, middleware provides some facilities, such as component models, to help the development and deployment of distributed systems. In a word, the proliferation means that middleware plays a much more important role in the execution of the whole system and has much more knowledge about the whole system.

In terms of the concepts and taxonomy built up by Avizienis et al. in [4], “the dependability of a system is the ability to avoid service failures that are more frequent or more severe than is acceptable”. The reliability and availability is two important aspect of dependability. In a technical view, a dependable system will monitor and measure, reason and plan, change and evaluate its states and behaviors at runtime for discovering or predicting and recovering or preventing system failures [18]. In middleware, the discovery of system failures and the corresponding recovery planning require enough details of the execution and plentiful knowledge of the semantics of the whole system. So, from the perspective of reliability and availability, the proliferation
of middleware means that it has to take much more responsibilities for detecting and recovering faults and failures than applications and underlying operating platforms.

The reliability and availability of middleware based systems can be supported by the application, the middleware and the underlying operating system respectively, as shown in Figure 1. For the application, it can detect the faults, errors and failures taking place at the application level and recover part of them because the application has and only has full control of the application code. The application can also detect some failures of the middleware and underlying operating system because it utilizes the underlying services and then has some knowledge of their failures. However, it is normally very hard, even impossible, for an application to recover the underlying failures because it has no control over the underlying middleware and operating system. For the underlying operating system, it cannot discover and recover the failures both at the application level and at middleware level because it has no or little knowledge of the semantics of the whole system. For middleware, it cannot discover or recover the faults and failures that are embedded in the application code or in the implementation of the operating system, or failures that have no manifestations. This is due to the fact that middleware controls applications’ execution via utilizing the capability of the operating system, under the guidance of its knowledge of the whole system. The framework presented in this paper can detect and recover the faults and failures taking place at the middleware layer in an automated way.

There are three ways to achieve fault-tolerant systems. The first one is to construct fault-free software. Researchers have traditionally relied on fault avoidance techniques, system validation, and analytical models to try to build fault-free software [13]. Decades of experience have led to numerous valuable techniques, which have improved the reliability and availability of software systems. But the accurate modeling and verification of complex computer systems and their environment are impractical in most cases [12]. The second way is replica-based fault tolerant technique. This kind of systems uses multiple replicates. When one replica fails, another replica would be used to deal with the user requests. There have been some important systems using object replication, such as Java RMI-based Arjuna [16], Microsoft DCOM-based COMERA [28], and CORBA-based OGS [29]. These systems can mask faults in most cases. However, replica-based fault tolerance has some critical shortcomings, such as more hardware is needed, performance overhead is significant and the cost of maintaining these hardware and systems is even higher. The third way is recovery based approach. It gains significant progress in recent years. The basic philosophy of recovery based fault-tolerance is: “since computer systems faults and operators error cannot be avoided in real world” [3], it would be better to improve system dependability by fast recovery than avoiding or masking faults by redundancy. Using recovery based approach, high reliability and availability can be achieved without using redundant hardware so that the cost of whole system, including the number of operators, can be lowered [21,7]. Recovery techniques have been widely used in hardware, operating systems and database management systems.

2.2 J2EE and Its Standard Services

Enterprises today need to extend their reach, reduce their costs, and lower their response times by providing easy-to-access services, which need to be highly available
and reliable, to their customers, partners, employees, and suppliers. The Java 2 platform enterprise edition (J2EE) defines a standard application programming model for developing multi-tier, thin-client application and a standard platform for hosting applications. J2EE reduces the cost and complexity of developing multi-tier services, resulting in services that can be rapidly deployed and easily enhanced as the enterprise responds to competitive pressures. There are plentiful standard services defined in J2EE\(^2\), as shown in Figure 2. Some services that suffer mutually dependent faults and will be studied in this paper are as follows:

- **Transaction service**: It transparently supports both programmatic transaction demarcation and declarative transaction demarcation to facilitate the assurance of ACID properties. The Java Transaction API (JTA) consists of two parts: An application-level demarcation interface that is used by the container and application components to demarcate transaction boundaries, and an interface between the transaction manager and a resource manager used at the J2EE SPI level (in a future release).

- **Data service**: It provides developers a standard way to use database and to manage database connection pools to gain better performance. Each data source corresponds to a pool. Connection-related operations, such as getting or closing a connection, are executed by pools, and concrete database operations, such as querying and updating, are performed by specific JDBC (Java Database Connectivity) driver classes provided by database product vendors.

- **Messaging service**: It is a standard API for messaging that supports reliable point-to-point messaging as well as the publish-subscribe model. A JMS provider must implement both point-to-point messaging and publish-subscribe messaging.

- **Naming service**: The Java Naming and Directory Interface (JNDI) API is the standard API for naming and directory access. The JNDI API has two parts: an application-level interface used by the application components to access naming and directory services and a service provider interface to attach a provider of a naming and directory service.
JAAS: The Java Authentication and Authorization Service (JAAS) enables services to authenticate and enforce access controls upon users. It implements a Java technology version of the standard Pluggable Authentication Module (PAM) framework, and extends the access control architecture of the Java 2 Platform in a compatible fashion to support user-based authorization.

It should be noted that new standard services will be added in the new version of J2EE specification. For example, JAAS, JAXP (Java API for XML Processing) andConnectors are added in J2EE version 1.3, and the services supporting web services are added in J2EE version 1.4. Since the number of J2EE standard services will increase continuously, the reliability and availability of J2EE services will put more and more significant impacts on the whole system and the problem addressed by this paper becomes very important and urgent in J2EE.

3 Mutually Dependent Faults

The invocations between middleware services make these services dependent on each other. This would lead to a chain of faults when a service failed. In this section, a detailed analysis about such a complex relationship in middleware services is presented.

3.1 Interdependent Relationship

When implementing a middleware or a middleware service, service providers usually reuse other existing services to fulfill the desired functionality. If a service uses another service, we say that the former service (depending service) depends on the latter service (depended-on service), and there is an interdependent relationship between these two services. For example, the transaction service depends on the data service. Figure 3 shows the sequence and interactions of transaction operations and database connection operations. The transaction service invokes the data service operations to commit a transaction. When a transaction is to be committed, it calls the commit operations of database connection corresponding to this transaction. So, we call the transaction service in the scenario the depending service and the data service depend-on service.

Figure 3. Sequence diagram of a transaction
The services’ interdependent relationships can be classified into three types according to the depending degree:

- The depending service initiates or acquires the depended-on service instance when it wants to use the depended-on service, and then calls the depended-on services’ methods. The interdependent relationships between the naming service and other services in J2EE are mostly of this type. Each time a service wants to use the naming service, it initiates a NamingContext instance and invokes its lookup method.

- The depending service initiates some depended-on service instances or acquires references and keeps the references to these instances until it does not use the depended-on services any more. When the depending service wants to use the depended-on service again, it calls methods of the instance references directly. The instance reference is kept, instead of initiating a new instance every time, not only because of simplicity but some initiating parameters and environmental variables are constant—it is only available at the startup time. In JOnAS, an open source J2EE application server, JMS requires a TransactionManager instance as one of its startup parameter and uses this instance to start a new transaction each time it sends a message to a message receiver. JMS can get TransactionManager instance only at the startup time.

- The depending service and the depended-on service are tightly tied up with each other with conversational states during several method invocations. For example, the interdependent relationship between the data service and the transaction service is of this type. After a transaction begins, the transaction service keeps a list of database operations that belong to the transaction. Such database operations are obviously information of the data service.

3.2 Mutually Dependent Faults and Correlated Failures

Normally, when a service fails, the application depending on the service cannot get desired functions from it and cannot provide correct functionality to other services or end users. The reason for the loss of functionality is simple and clear outside of the application but becomes complex and vague when investigating the internal and root causes due to the complex interdependent relationships between middleware services. Mutually dependent faults \[22\] are induced by interdependent services. If the depended-on service fails, the depending service cannot provide correct functions because of the failures of depended-on service. The failures of depended-on service and depending service are tied up with each other and hence called correlated failures. Again, we take the data service and the transaction service for example (refer to Figure 3). In a real world, there are many causes of the data service failure, such as network errors and database failures. When failures occur, the database connections gotten before are no longer valid and any operation on them will fail. Then, when transactions depending on these invalid connections are committed or rolled back, failures will also occur. For example, if a network error occurs after the Operation (3), when the Operation (4) commits the transaction, the database operations must be committed first. But the database connection is invalid at that time, so the Operation
(5) will fail, too. In such a case, the failures of the data service and the transaction service are correlated.

Our experiment also proves the interdependent relationship of faults in the data service and the transaction service. We use ECperf, an enterprise benchmark application, as the test application and stop the database for 10 seconds in every 60 seconds to trigger the data service failures. Figure 4 shows the experiment result on PKUAS. It is clear that when the database stops, i.e. the data service fails, some transactions fail either.

3.3 Recovering Correlated Failures

Naturally, service failures can be recovered in a simple detect-recover manner: a service will be recovered if its failures are detected. In such a manner, whether the failures are of the same service is not important: if several failures are detected in more than one service, all these services are recovered. If the failures are not correlated, they can be recovered independently. But in the case of correlated failures, the recovery of depended and depending services must be coordinated. Because recovering the depended-on service and depending service separately may leads to low efficiency. A longer recovery time means a larger MTTR. This will definitely decrease middleware’s availability. In a worst case, only when the depending fault is recovered can the affected service be recovered successfully.

Corresponding to the classification of service interdependent relationship, the coordinated recovery can also be classified into three types:

- Only recovering the depended-on service. In the first type, there is no association between the usages of depended-on service. That is to say, each time the depending service calls the depended-on service, the request will be fulfilled if there are no faults and errors in the depended-on service.

- The reference of the depended-on service held by the depending service has to be updated. After the recovery of depended-on services, the references kept in the depending service have great possibility to be invalid which may cause
Hong Mei, et al.: Coordinated recovery of middleware services: A framework ...

depending service failed. If we consider updating references as the recovery of the depending service, the key of the coordination is to guarantee the execution order of the two recovery processes.

- The recovery of interdependent services must consider the states and information of the other services. This means the recovery operations have interdependent relationships, too. So the implementation of recovery operations is more complex. And furthermore, the recovery of each service may include several sub-processes and these sub-processes may need to be coordinated to execute in a specific or interweaving order.

Comparing to the independent recovery of interdependent services, the coordinated recovery can improve both the performance and efficiency of recovery. In the first type, only one recovering process is needed. In the second type, if the recovery is not coordinated and the recovery process of the depending service is performed earlier than that of the depended-on service; the recovery of the depending service will have no effect. In the last type, such complex recovery process cannot even be carried out without coordination.

4 Framework Overview

Based on the analysis above, we give a basic coordinated recovery model to deal with mutually dependent faults and correlated failures in this section. In addition, several critical supporting mechanisms in the model are described in detail, including JARME to discover correlated failures, a set of exception-catching based fault detectors to monitor abnormal behaviors, some guidelines to obtain recoverable services, catching and redirecting mechanism to provide transparent fault recovery, and a recovery policy to combine error detection and fault recovery activities.

4.1 Coordinated Recovery Model

In terms of the above investigation of mutually dependent faults and correlated failures in middleware services, we define a general model for coordinated recovery of the failures, as show in Figure 5. The model consists of a set of Fault Detectors, some recoverable services, and a Recovery Coordinator with Recovery Policy.

![Coordinated Recovery Model Diagram](image-url)
4.1.1 Fault detector

Middleware services may fail in many cases and have different phenomena, so single fault detector cannot detect all kinds of faults. As a result, a set of fault detectors is necessary. They monitor the services’ behavior, gather runtime information and send out alerts when faults occur.

There are many mature techniques, such as exception tracing and statistic analysis of historical data, to detect faults. Exception tracing is based on the exception mechanism of programming languages and the fact that a given exception is raised if a failure occurs in middleware services. Because an exception is raised at the time the failure occurs, a detector using exception tracing can react quickly. By analyzing the exception stack and the messages attached on the exception, the root cause of faults may also be found. Another fault detector, statistic analysis of historical data, requires recording runtime data and analyzing these data periodically. The basic idea of this approach is that unusual changes of data may indicate that some failures occur.  

4.1.2 Recoverable service

Middleware service has the best knowledge of itself. So it is reasonable to recover service failures by the service itself. A recoverable service divides itself into some recovery units and provides a set of recovery operations to recover a subset of the units or all of them. These recovery operations are passive, or, in other words, they are taken when an external commander, the Recovery Coordinator, requires them to do so. The recoverable service is an extension to the original service so that the service can be recovered and the recovery operations can be coordinated by the Recovery Coordinator. Each recoverable service has some common features and optional features. The common features are common recovery strategies which are effective to most services, such as service reboot and internal state persistency. The optional features are service-specific recovery strategies which make the recovery procedure more efficient or lower the recovery cost. More details of making a service recoverable will be discussed later.  

4.1.3 Recovery coordinator and recovery policy

Recovery Coordinator is a central controller to coordinate recovery operations of recoverable services according to Recovery Policy. Recovery Policy, which will be discussed in the following section, defines the faults to be monitored and the way how to recover. Recovery Coordinator has two main responsibilities. First, it is responsible for deciding when and how to recover failures. After receiving alerts sent by one or more fault detectors, Recovery Coordinator analyzes the alert messages to find out the root services of the faults. Then according to the analysis result, it executes recovery operations according to the corresponding Recovery Policy. Second, Recovery Coordinator guarantees that the chosen recovery operation is not only successfully performed but also executed in the right order.  

4.2 Supporting Framework

Though the coordinated recovery model is general enough, many specific mechanisms are required when applying the model to a given middleware. Here, we will discuss how to support the model in J2EE, including discovering mutually dependent faults by fault injection, detecting the occurrence of faults by exception, making middleware services recoverable, caching incoming requests when a failure is being
handled, specifying the behavior of the framework by recovery policies, and so on. It should be noted that all mechanisms discussed here are implemented in three different J2EE application servers, i.e., PKUAS, JBoss and JOnAS. In this sense, the framework is general enough for improving the reliability and availability of J2EE systems by recovering correlated failures.

4.2.1 Discovering correlated failures

Failures can be detected and recovered only after they are defined explicitly and clearly. This is a hard work because there are many service failures in middleware and each of them should be analyzed to see whether it can cause faults of other services. Reviewing service implementation documents and source codes is one way to discover such failures. The mutually dependent faults are hard to discover manually and it is very likely that some will be missed. Besides, this work is obviously boring, time-consuming and error-prone. However, most services use other services via standard service API and API-SPI (Application Programming Interface - Service Provider Interface) pattern which provides application programmers a set of API while requires service providers to implement a set of SPI different to API. In our solution, we use software implemented fault injection to simulate middleware service failures and observe the reactions of other services. This automatic, experimental approach, along with document/source code reviewing and experts’ experience, discover most of mutually dependent faults.

4.2.1.1 Overview of JARME

Fault injection is a technique to observe a system’s behavior when a special input, i.e., faults, is introduced into the system [2]. By accelerating the occurrence of faults, errors and failures, fault injection has been widely used in validating the dependability properties of a system and evaluating the impact of error detection/recovery mechanisms on a system’s performance [9]. Normally, fault injection can be implemented by hardware or software. Hardware implemented fault injection (HWIFI) focuses on causing actual hardware faults which may be close to a realistic fault model. But this kind of fault injection needs specifically designed hardware to enable the injection of faults so it is expensive and dedicated to a specific target system. In addition, the consequent software faults caused by HWIFI cannot be well controlled and estimated. On the contrary, software implemented fault injection (SWIFI) emulates the error state of the system hardware and software through special programs. It is a cost-effective and efficient method without using extra hardware and is able to be easily expanded for new fault types. Moreover, SWIFI can be accurately controlled to decide when, where and what to inject.

We propose a system reliability measuring environment (J2EE Application Reliability Measuring Environment, JARME) which focuses on measuring the impact of middleware service faults. In this environment, service faults are simulated by injecting Java exceptions into services at runtime. JARME’s architecture is shown in Figure 6. JARME works on a J2EE application server, i.e. the Integrated Application Server in Figure 6, which can provide the runtime environment for Business Applications. The Business Application is the target application under testing, and Use Case Based Test Clients are programs invoking server-side methods to perform a defined test case. Server Initiator is responsible for adding failure injection codes into services implementation of Integrated Application Server when services are loaded.
and controlling the start and stop of application server at runtime. This dynamic AOP approach makes application server code separate from fault injection code and makes the original application server code intact during its normal execution. Service Failure Injector triggers the injection of service failure by invoking fault-triggering methods that are 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. Framework Controller coordinates Server Initiator, Service Failure Injector and Test Client Controller.

![Figure 6. JARME architecture](image)

### 4.2.1.2 Discovering correlated failures by JARME

The ability to simulate service failures using exceptions in JARME can greatly help to discover correlated failures. For a given application and a test case, the process of discovering correlated failures is:

1. Configuring JARME to simulate specific exceptions of a given service and the frequency and duration of the exceptions;
2. Configuring test client to simulate single end user. The simulated end user will perform the test case;
3. JARME generates exception-thrown codes and controller codes in the form of AOP aspects according to above configurations;
4. Framework Controller starts the test process by invoking Server Initiator to start Integrated Application Server and insert exception-thrown code into the service;
5. After the application server is started and Business Application is deployed, Test Client Controller invokes the test client to perform the test case;
6. Service Failure Injector activates the exception that are inserted into service according to given frequency and duration;
7. Framework Controller logs every service exception;
8. JARME analysis the log to find how the configured exception in JARME cause other exceptions of other services, and these exceptions are correlated failures;
9. Going back to step (1) to simulate failures of another service until failures of all services are simulated.

In each round of the process, failures of one specific service are simulated. And if the test client invokes application when faults are injected, it will get an exception which contains the information of invoking chain and the root exception. In step (8), all exception tracing information is analyzed. If the invoking chain contains service other than the configured one, correlated failures are discovered. Then the discovered correlated failures are defined using the exceptions thrown in the interdependent services. The process is a repeated one that in each round faults are injected into one
service. After the injection involves all service, most correlated failures relating to this test case are discovered.

However, JARME cannot discover all correlated failures by now because of the following three causes. First, the discovered correlated failures are specific to a given application and a given scenario, so the coverage of JARME is decided by the coverage of test cases in study. This shortcoming comes from fault injection technique. Second, if there are latent faults in the test scenario, JARME cannot discover the correlated failures, i.e. JARME cannot discover the failures that are not be triggered in a test case. This is quite enough in most cases because the missing correlated failures will not impact system reliability since they cannot be triggered when the system is running. For example, the transaction service and data service have correlated failures definitely, but if an application does not use the transaction service at all, the correlated failures will not occur in this application and it is unnecessary to detect and recover such failures in this application. Finally, not all failures throw exceptions in J2EE application server. Java exception would be a representative sign of failures in about 60% of failures in our study. Another study already showed that Java exception can cover most of environmental failures at runtime. Despite the shortcoming, we still believe JARME is a usable and automatic approach to simulate most of faults in an application server, and hence can discover major correlated failures. In addition, we use only one test client in correlated failures discovery now because more test clients would import complex but sometimes fake dependencies. For example, when two test clients execute simultaneously and trigger independent faults concurrently, the corresponding exception traces would be mixed together in log file. This will lead to unnecessary difficulties to resolve the log correctly. Other mechanisms, such as Machine Learning on system behaviors, are needed to deal with above two problems and we will take it to improve fault coverage.

4.2.2 Designing fault detectors

The classification of mutually dependent faults helps to design the fault detectors (See 3.2). From the classification, we can see that in the second type of mutually dependent faults, the faults of depended-on service can be detected only by the depending service. So we should design corresponding fault detectors which are placed in the depending service and take the responsibility of detecting faults of the depended-on service.

The design pattern, Interception, is used to design flexible and reusable fault detectors. If the fault detectors act as interceptors, they can intercept each request and analyze the return values to see whether the service provides correct functions. If there are several fault detectors concerning the same operation of a service, they can be organized as a detector chain.

As mentioned above, there are many techniques to monitor service behavior and detect middleware faults. Among these techniques, exception tracing is easy to use and encouraged in our model. One reason is that exceptions are supported by programming languages very well and are widely used to denote unexpected situations during system execution. Exception tracing provides an easy way to analyze the faults in detail because exceptions are thrown in an inverse order of service invocations. The top exception in an exception trace stack is the root cause of fault. Another reason is that exception tracing can simplify the implementation of fault detectors which are
designed as interceptor. The simplest way to implement a fault detector is to add a try-catch block around the real service method and analyzing methods to ascertain whether the caught exceptions denote the faults concerned by this detector.

4.2.3  Making services recoverable

If we add some recovery features into a service so that the service can be recovered under the coordination of the recover coordinator, the service is recoverable. The features may be common, which are effective to most services, or be specific, which must be implemented specially and are only effective in some services.

4.2.3.1  Common features

The common features include service rebooting and reference updating. Rebooting is a widely used recovery techniques. When failures are detected in a service, the service can be recovered by rebooting. Rebooting can recover a majority of failures, especially those caused by transient faults. Furthermore, rebooting is easy to implement. After rebooting a service, the references to its instance or its component instances may be no longer valid. Then the depending service, which possibly keeps these references, should update its references. So reference updating feature should also be provided. A typical scenario of using both service reboot and reference updating is that rebooting the depended-on service and updating the references in the depending service.

4.2.3.2  Optional features

The optional features include the features that are different either in the implementation or the recovery strategies. Because there are many optional features, we only list some examples here.

- Check point: Check point is the time when a service checks all its inner states and stores them in a persistent storage. Check points are set along the execution time line of a service. When faults occur, the middleware can restore service’s states from the nearest check point. The check point helps the quick recovery without losing service’s states.

- Caching request: When a failure is detected, subsequent requests for the service should be cached for a period of time until the service is recovered. Then these requests can be fulfilled properly. Caching request itself cannot recover a service, but it helps to minimize the influence of failures and provides transparent recovery. The order of several requests issued by a user need not be considered because a user’s request obeys request-reply pattern in a J2EE application. This means a user would not send the next request unless the former response is received.

- Redoing request: If a failure is detected, the requests under execution can be canceled at that time and re-executed after the failure is recovered. The redoing request can help to minimize the failed requests and hence improve user availability except delay, because some requests are executed again after the service is recovered and returned correctly. Remembering request-reply pattern in a J2EE application, we need not to take into account the relationship between cached requests and new arrived requests.

4.2.3.3  Implementing recoverable services
Our framework is implemented in PKUAS, JBoss and JOnAS. Because of the complex semantic dependency between different services in different application servers, we have to make different service implementations recoverable by hand. However, the basic idea of modification of a service is common in different application servers in spite of some specific code modifications. The services that are made to be recoverable according to the above common and optional features include:

- **Recoverable JAAS service**: Java authentication and authorization service (JAAS) \(^{[26]}\) provides means by which communicating entities (for example, client and server) prove to one another that they are acting on behalf of specific identities that are authorized for access, and means by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints. If a failure occurs in JAAS, it can be recovered simply by restarting. Because the authentication and authorization information such as username-password and access control list are stored in a persistent storage such as file system and database, no inner state or information is needed to be kept. The recovery of JAAS stands for the simplest cases of middleware service recovery.

- **Recoverable naming service**: The Java Naming and Directory Interface (JNDI) API \(^{[27]}\) provided by naming service is the standard API for naming and directory access. The JNDI API provides the component standard way to register itself to the naming service so that others can find it by looking up the component’s name in the naming service. At runtime, the naming service maintains a table of registered components after the form of name-reference pair. To avoid the losing of the table when a failure occurred and during recovering, recoverable naming service stores the table to a persistent storage at the check point and the check point is set periodically. When failures are detected in the naming service, the failures can be recovered by rebooting the naming service and restore the registry table from the persistent storage.

- **Recoverable Java Message Service**: The Java Message Service (JMS) \(^{[28]}\) is a standard API for messaging that supports reliable point-to-point messaging as well as the publish-subscribe model. The recovery of JMS is similar to that of naming service. The only difference is that the information needs to be stored includes the message queues, topics and the registered message listeners. Furthermore, the JMS keeps a reference of transaction manager. If the transaction service is recovered, the reference has a great possibility to be invalid. So when recovering the correlated failures with transaction service, the reference of transaction manager should be updated also.

- **Recoverable data service**: The main idea of recovery of the data service is: when a database connection in pool is found invalid, all connections in the pool are checked and repaired. The basic recovery unit is a wrapper of actual database connection. Any connection that the user gets from the connection pool is a wrapper connection, which holds a reference of actual connection and takes the responsibility of detecting and recovering failures. The actual connection’s
reference is used to execute the database operations, i.e. querying and updating. When a connection becomes invalid and is detected by Fault detector, DB_CONNECT_FAIL alert would be sent out. By getting a new and valid database connection and updating the reference of actual database connection, the wrapper connection recovers itself and the recovery process is transparent to end users. The database connection pool as a whole is a bigger recovery unit which recovers all connections in the pool. Indeed, each recovery unit is a combination of fault detector and failure repairer. To help the recoverable transaction service to redo transactional operations, the data service also makes its statement and prepared statement having the ability of redoing the operations on the repaired JDBC connections.

- Recoverable transaction service: Logging user operations and automatically redoing these operations when failures occur is a widely used technique in fault-tolerant and self-recovery systems. The recovery of the transaction service is based on this technique. And 3R Model [6] of ROC (A philosophy of emphasizing on recovery from failures rather than purely failure-avoidance. More details of ROC would be presented in section 6), is also referred to: After a transaction begins, any operation of the transaction is logged until it ends; and a transaction is recovered by redoing every operation which has been logged. Considering the speed of different mediums including memory, file system and database, we choose memory as logging medium. The other reasons include the recovery of transaction service concerns the situation that middleware is running on a single machine and the middleware is still working. If the memory fails, middleware must fail too. In such a case, it is impossible to recover any services. In addition, the logging data includes only the ongoing transactional operations, so it would not require a large memory space. It is obvious to choose the beginning of a transaction as a checkpoint because it is the beginning state of redo operations. Any transactional operation is performed permanently on database at the time of the transaction committing, so the operations on current invalid database connection has no influence on the data in the database. And hence, from the newly created database connection’s point of view, the data in the database is exactly consistent with the state of the beginning of transaction. And the data independency and consistency of transaction can be guaranteed. When transactions are re-done, there may be conflict among transactional operations. In such a situation, the database decides which transaction to be committed, and others will be rolled back. So operations’ transactional attributes are still guaranteed. The extension of original implementation class of single transaction logs each transactional operation and also acts as a fault detector which monitors the beginning, committing and rolling back of the transaction. Whenever a failure of transactional operation is detected, RTXCapsule tries to find out the root cause using exception tracing technique. If the analysis result indicates that some kinds of database errors cause this failure, a TX_DB_CAUSE_FAIL alert will be sent out.

4.2.4 Caching requests
Caching request can lower the failure damage and can be implemented at the
system level instead of at the service level. In middleware, the thread pool is usually used to minimize the performance overhead of creating and releasing threads. A thread is put back to the pool, instead of being released after finishing its execution. The requests are placed in a queue, waiting for free threads. The caching can be implemented by stopping the assignment of free threads to the waiting requests so that the requests will wait until the recovery finishes.

In our implementation, we implement the caching at the system level, whose benefit is that any request can be cached and the caching mechanism need not be implemented in every service separately. However the system level caching will cache all requests including requests that do not use the failed services. This will induce unnecessary performance impact indeed. But we believe this is a safe strategy because we cannot distinguish requests that will use the failed services from that will not use the failed services now. Caching all requests after detecting a fault would not cause extra failures during recovery. Furthermore, the performance penalty of catching all requests can be ignored in our quick restarting context.

4.2.5 Recovery policy

Recovery policy is the basis of coordinator’s actions. We use an independent configuration file for making the recovery more flexible, customizable and scalable. Recovery policy is an XML file that defines possible service faults and their corresponding recovery process. To add a new kind of fault or a new recovery process, the only activity is to modify recovery policy file. Recovery policy can be divided into two parts: the definition of correlated failures and the definition of recovery process. Recall the classification of mutually dependent faults, this separation helps to separate the fault detection and failure recovery.

The definition of correlated failures is generated from the analysis results of JARME. In our framework, fault detector identifies single fault and sends out an alert message. In the message, an identifying fault name is set to indicate the fault type. And the correlated failure is defined as a set of middleware faults. Furthermore, because the correlated failures occur when the client calls a service and this service calls another service, these failures must occur in the same thread. So the coordinator determines the occurrence of correlated failures if and only if the failures defined in the correlated failures occur in the same thread.

The definition of recovery process defines all necessary recovery operations to recover correlated failures. As discussed above, the recovery of correlated failures of interdependent services may involve more than one service. In such cases, a recovery process consists of recovery operations of more than one service. And considering the interdependent relationships among these services, these operations must be executed in a given order, so these operations must be arranged in the given order. When the Recovery Policy file is being parsed, the order of operations defined in the file is the same as the order in which the operations enter the parser. The Recovery Coordinator must ensure the success of prior recovery operations before starting the latter ones. If the prior ones fail, the latter will be held up, because otherwise these operations will definitely fail also. The attempt to recover depended-on services is carried out periodically.

Figure 7(a) shows an example of recovery policy used in the recovery of the data service and the transaction service in PKUAS. In the given policies, only one corre-
lated failure is defined. When DB\_CONNECT\_FAIL alert and TX\_DB\_CAUSE\_FAIL alert are sent in series, Recovery Process 1 will be carried out. The definition of Recovery Process 1 includes the recovery operations of the data service and the transaction service which recover all database connections and all transactions whenever the correlated failures are detected. Recovery Process 1 also instructs that Recovery Coordinator must perform the data service’s recovery operation, ahead of the transaction service’s redoTx operation.

Recovery policy can also be used in defining uncorrelated failure and its recovery process. In such a case, a correlated failures definition in fact contains only one failure and a recovery process definition contains only one recovery step with one recovery operation. Figure 7(b) shows an example of recovery policy for recovering uncorrelated failure in the naming service.

![Recovery Policy for Correlated Failures](image1.png)

![Recovery Policy for Uncorrelated Failures](image2.png)

Figure 7. Samples of recover policy

5 Experimental Validation

In this section, we carry out a number of controlled experiments to validate the effectiveness and efficiency of the framework. The sample application is ECperf, an EJB benchmark to measure the scalability and performance of J2EE servers and containers. It contains four domains - Manufacturing, Supplier & Service Provider, Customer, and Corporate - to simulate the process of manufacturing, supply chain management, and order/inventory of the business problem. Data service, transaction
service and JAAS are critical in such a business application. At the beginning of
the experiments, we employ JARME to discover some correlated failures using some
test cases. Also, JARME helps to validate some of our estimates about correlated
failures. The correlated failures include the data service and the transaction service,
JNDI and JAAS, and JMS and the transaction service. Then, based on these findings,
we define recovery policy files as input for Recovery Coordinator. We will emphasize
coordinated recovery framework in this section, to validate the efficiency of the coordi-
nated recovery framework. To illustrate the generalization of the framework, all cases
are studied in all of the three application servers, i.e., PKUAS, JBoss and JOnAS.
But for keeping it simple, we will discuss three different correlated failures and their
recovery results in different application servers. Because the recovery of correlated
failures of the data service and the transaction service is the most complex in
our experiments, it will be discussed in detail. Other cases are less complex and are
easy to understand, so we only briefly introduce their interdependent relationships
and the recovery strategies, and show experimental results. At last, we will discuss
the performance penalty of the framework.

5.1 Case I: Recovery of the Data Service and the Transaction Service

To validate the effectiveness of the coordinated recovery of the data service and
the transaction service to improve application server’s reliability and availability, we
use a typical ECperf scenario to get the comparative test data. In the scenario,
a customer lists and selects some products, and then sends an order to a product
provider. The data service and transaction service are the keys to ensure successful
ordering in this scenario. To test the effectiveness of coordinated recovery, a separate
process stops the database periodically when the application server is processing the
customer’s request.

At the server side, four PKUAS instances are used as shown in Table 1. PKUAS
I only recover the data service, without considering transaction. If the data service
finds a JDBC connection in the connection pool fails to perform operations, this
connection is abandoned and new connection is gotten and put into the connection
pool. PKUAS II provides request caching when the data service fails, besides the
recoverable data service in PKUAS I. Any request to the connection pool is cached
until the data service recovers itself. PKUAS III and PKUAS IV use the coordinated
recovery described in previous sections. But PKUAS III does not use request caching.

<table>
<thead>
<tr>
<th>Instances</th>
<th>Caching</th>
<th>Coordinated Recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>II</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>III</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>IV</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

At the client side, an open source performance test tool, OpenSTA \(^{[20]}\), is used
to simulate concurrent users to send requests to ECperf and collect the test data.
OpenSTA simulates concurrent users using multiple threads and requests are sent
sequentially in each thread: one request is sent after the previous request’s response
is received. If the response shows that the request has been fulfilled successfully, the
successfully completion count is incremented by one; otherwise the failed completion count is incremented by one. The counting time is the time when the test client receives responses.

Both PKUAS with the deployed ECperf and test client are running on PC (Intel PIV 2.8GHz, 512M Memory, Windows XP with Service Pack 1). In the case, a separate process stops the database 10 seconds every 60 seconds after the test application runs for 1 minute to simulate transient database failure and intermittent network disconnecting. (We use such a fault load because 10 seconds is a suitable delay to stop a database and 50 seconds is long enough to restart the database and resume the normal process with steady request response time.). The test lasts 400 seconds. Figure 8 and Table 2 show the experiment results. In the four PKUAS instances, there are altogether six troughs and the number of troughs exactly matches the number of times the database has stopped.

![Figure 8. User requests completing ratio](image)

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Failed transaction count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PKUAS I</td>
</tr>
<tr>
<td>Overall Transaction</td>
<td>42351</td>
</tr>
<tr>
<td>Failed Transaction</td>
<td>3756</td>
</tr>
<tr>
<td>Availability</td>
<td>91.13%</td>
</tr>
</tbody>
</table>

PKUAS I only uses the recovery of data service. When the database stops, about a half of the user transactions fails, and after the database starts again most of user transactions succeed. This indicates that the recoverable data service can recover itself after the database works again, but it cannot prevent the failure of user transactions when the database fails. So, the number of successfully committed transactions goes down when the database stops.

PKUAS II uses both the recovery of data service and request caching. After the database fails, operations of newly started transactions are cached until the data service is recovered. The benefit of request caching is observably shown in Table 2 that the percentage of failed transaction PKUAS II (0.62%) is much smaller than that of PKUAS I (8.87%). This identifies that request caching is very important in failure recovery.
After using the coordinated recovery of the data service and transaction service in PKUAS III, the successfully committed transactions ratio raises much more. Although no cache is used, there are still troughs. This is because when the database stops, any operation related to the database has to wait until connections are timeout. So the count of the requests performed decreases when the database stops. Without request caching, requests are performed as normal when database stops, instead of waiting. But the recovery process can be carried out only if the database starts again. This means these requests have ended before recovery. So these requests cannot be successfully performed and recovered, the count of failed transaction is 121.

After using request caching along with coordinated recovery of the data service and the transaction service in PKUAS IV, almost all user transactions are successfully committed. The successfully committed ratio, which is shown in Table 2, is up to 99.97%. In Table 2, the failed transaction counts of PKUAS III and PKUAS IV are lower than that of PKUAS II. This is due to the coordinated recovery of the data service and the transaction services. In PKUAS II, if the data service fails, all started transactions have no other choice but rolling back. The data service has the recovery ability and it can recover the JDBC connection. But the transaction is tightly connected with some specific JDBC connections and the committing of transactions depends on the committing of these JDBC connections. If these connections fail, the operations performed on these connections are not committed. Furthermore, without coordination, the transactions depends on these failed connections cannot redo operations that have been performed before. So when the data service fails, the transactions which were started will fail definitely. In PKUAS III and PKUAS IV, under the coordination of Recovery Coordinator, the transaction service can redo previous operations which were performed before and not committed at the time the data service fail. In other words, it is the coordination that decreases the number of failed transactions.

The effect of request caching should not be ignored. Using request caching, user requests are held so that they have the opportunity to be recovered. As we can see from Table 2, the availability of PKUAS II is much higher than that of PKUAS I, and the increase of availability is just contributed to request caching. It is also the same to the increase of availability of PKUAS IV comparing with that of PKUAS III.

5.2 Case II: Recovery of JNDI and JAAS

To validate the high-efficiency of coordinated recovery compare to non-coordinated recovery, we carry out case II and case III. A higher efficiency of recovery means less MTTR and as a result, higher reliability and availability.

Besides providing fundamental naming functionalities, the JNDI service in J2EE also considers security concerns. The JNDI service can install a security manager while running. The ability of an applet or an application running with a security manager installed to access service providers, especially service providers that require the use of restricted resources (like the file system or network connections) may be severely limited. If a security manager is installed and a request is received, the JNDI service will do the authentication before providing the service functions. This process is implemented as a sequence of initiating a security manager of JAAS and calling proper methods, i.e. login, of it. So the JNDI service depends on the JAAS service.
which leads to the correlated failures: if JAAS failed, obviously this process will fail either and the consequence is a JNDI failure.

Regardless the interdependent relationships, these two services can be recovered by roll-back recovery or direct reboot, as stated in section 4.2.3.3. But seeing that the recovery of the correlated failures in JAAS and JNDI is a typical case of the first type of recovery of correlated failures discussed before, the coordinated recovery can also be applied here. If JAAS is recovered successfully, the next time JNDI successfully uses JAAS and there is no recovery needed in JNDI.

We do comparative experiments on JBoss by injecting JAAS specific exceptions. In the experiments, we modify the source codes of JBoss so that both the JAAS service and the JNDI service are recoverable using reboot. Two JBoss instances are configured: one is recovered without coordination (JBoss I) and one with coordination (JBoss II). In the experiments, a test client repeatedly executes a sequence of operations including getting NamingContext and looking up some object reference. Each round of test lasts 60 seconds, and JAAS failures are injected every five seconds from the 5 second. Both of JBoss instances detect the failures and recover them successfully, but differ on recovery performance, i.e., how fast the recovery is. Detailed recovery performance, which can be defined directly using the time used to recover service failures, is shown in Table 3. It is clearly shown that JBoss II can recover faster than JBoss I and the average recovery performance of JBoss II is 20% better than that of JBoss I. According to the philosophy of ROC, faster recovery means higher reliability. So the coordinated recovery improves system reliability.

<table>
<thead>
<tr>
<th>Injecting Time</th>
<th>5''</th>
<th>10''</th>
<th>15''</th>
<th>20''</th>
<th>25''</th>
<th>30''</th>
<th>35''</th>
<th>40''</th>
<th>45''</th>
<th>50''</th>
<th>55''</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>JBoss I</td>
<td>172</td>
<td>125</td>
<td>125</td>
<td>125</td>
<td>110</td>
<td>140</td>
<td>125</td>
<td>140</td>
<td>125</td>
<td>125</td>
<td>109</td>
<td>117.8</td>
</tr>
<tr>
<td>JBoss II</td>
<td>94</td>
<td>109</td>
<td>110</td>
<td>110</td>
<td>109</td>
<td>109</td>
<td>109</td>
<td>109</td>
<td>94</td>
<td>109</td>
<td>94</td>
<td>95.1</td>
</tr>
</tbody>
</table>

5.3 Case III: Recovery of Java Message Service and the Transaction Service

In the JMS specification, it is an optional requirement that the JMS supports distributed transactions via Java Transaction API (JTA). This optional requirement is implemented by keeping a reference of the transaction manager and some references of other components of the transaction service in the JMS implementation, and these references are used to begin, commit or rollback transactions when necessary.

If failures occur in the transaction service, the operations on the transaction manager will definitely fail. So the failure of the transaction service and those of JMS are dependent. We define the correlated failures as a combination of TX_FAULT and JMS_TX_FAULT. The TX_fault is the fault of the transaction service itself which is different to the fault TX_DB_CAUSE_FAIL. And the JMS_TX_FAULT is the fault caused by the failures of the transaction service. This definition differentiates the correlated failures of JMS and the transaction service from the correlated failures of the data service and the transaction service.

To recover the correlated failures, the transaction service must be recovered first. And then, even if the transaction service is recovered after the failures occur, the references kept by the JMS implementation are mostly invalid. So the JMS needs perform another recovery process to update the references. The recovery strategies
used in the recovery implementation have been discussed in section 4.2.3.3. Furthermore, redoing requests can be applied here, i.e. after both the JMS service and the transaction service are recovered, the failed request is redone so that the end user can be masked from the service failures.

The contrasting experiments are done on JOnAS by injecting transaction exceptions. We use three JOnAS instances. The first JOnAS instance (JOnAS I) uses only independent recovery and the JMS and the transaction service are recovered separately. The second JOnAS instance (JOnAS II) coordinated recovers these two services as stated in section 4.2.3.3. And the third JOnAS instance adds redoing request into JOnAS II. Request caching is used in all JOnAS instances. In the experiments, a message driven bean is used to consume the messages sent by a client who sends a message every 10 milliseconds. A round of the test lasts for 60 seconds and the failures are injected into the transaction service every 5 seconds. The experimental results show that the recovery performance of JOnAS II and JOnAS III is about 50% better than that of JOnAS I, as shown in Table 4. This validates that the coordinated recovery can improve the recovery performance and system reliability. Because any request sent when failures are detected is cached until the recovering finishes, the failed request count of JOnAS I and JOnAS II is both 1. And furthermore, after the using of redoing request in JOnAS III, there is no failed request from the end user’s perspective.

<table>
<thead>
<tr>
<th>Injecting Time</th>
<th>5''</th>
<th>10''</th>
<th>15''</th>
<th>20''</th>
<th>25''</th>
<th>30''</th>
<th>35''</th>
<th>40''</th>
<th>45''</th>
<th>50''</th>
<th>55''</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOnAS I</td>
<td>31</td>
<td>32</td>
<td>31</td>
<td>31</td>
<td>32</td>
<td>31</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>31.6</td>
</tr>
<tr>
<td>JOnAS II</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>15</td>
<td>15</td>
<td>16</td>
<td>16</td>
<td>15</td>
<td>16</td>
<td>16</td>
<td>15</td>
<td>15.6</td>
</tr>
<tr>
<td>JOnAS III</td>
<td>17</td>
<td>16</td>
<td>16</td>
<td>15</td>
<td>17</td>
<td>15</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>17</td>
<td>17</td>
<td>16.1</td>
</tr>
</tbody>
</table>

5.4 Performance Evaluation

We take the recovery of correlated failures of the data service and the transaction service as a performance evaluation example. In the comparative test without stopping the database, the average response time of PKUAS I and PKUAS II is 0.165 second, and that of PKUAS III and PKUAS IV is 0.170 second. We believe this minor performance penalty is acceptable, compared to a higher reliability and availability. The recovery of the data service and the transaction service is the most complex case, because the recovery involving operations of both services and fault detectors are set in both services. In simpler cases, the performance impact of the coordinated framework will be less. Other cases on JBoss and JOnAS have the similar performance impact.

In fact, the performance cost of the framework is mainly from the fault detectors and recovery operations. In our previous work [34,35], monitoring every invocation in PKUAS puts a little impact on the response time and throughput of the whole system. The fault detectors monitor part of the invocations and do some simple analysis, which will not impact the system performance significantly. Some recovery operations may have a significant impact if they are very complex. For example, it takes a relatively long time to reboot a JMS server. However, such impact is acceptable because it only takes place when failures occur.
6 Related Work and Discussion

There is some work that addresses the problem of correlated failures and correlated failures. Yuan-Shun Dai et al. propose a software reliability model framework based on Markov renewal model which has the capability of modeling the dependencies among successive software runs and can deal with multiple types of failures \[31\]. The main advantage of their work is to consider the correlation of successive software runs. But the model is quite complex.

The coordinated recovery has also been noticed by others, such as BMC Software provides the coordinated recovery of IMS and DB2 products \[5\]. But their solution is both product-specific and service-specific, and what we want to do is to provide a general framework suitable for most middleware services. Yansong Ren et al \[30\] focus on the dependable initialization of large-scale distributed software and they use a central coordinator to coordinate the recovery of interdependent initializing tasks. They divide initialization dependencies which identify the initializing tasks’ executing sequence into sequential dependency and operational dependency, and consider these dependencies in the recovery process if some tasks fail when initializing. And considering these dependencies, the recovery can be accomplished faster. Their work has similar motivation and philosophy to ours. The interdependent entities are initializing tasks and at the initializing time there is no user operation and the failures can be recovered by simply re-executing the tasks. When focusing on runtime faults of middleware services, we must consider the user operations and the re-executing of user requests cannot recover service faults. So we cannot apply their work directly to middleware service. But the rationale of considering interdependent relationship is quite important.

Recently, Recovery Oriented Computing (ROC) becomes a hot topic. The key techniques of ROC include fault injection and automatically fault diagnoses, fine-grain system partition and recursive micro-reboot \[7\], 3R Model \[6\], and Crash-only Software technique \[12\]. JAGR \[11\] applies ROC techniques into middleware, and extends JBoss \[14\] to recover faults. The extensions include: using fault injection technique to construct fault propagation map; adding monitors to detect faults; and rebooting specific EJB container according to fault propagation map when fault occurs. However the shortcoming of ROC is that user operations during system failing cannot be recovered. Also, JARG does not consider the influence of service’s failures and recovers failures only by reboot. In our framework, correlated failures are addressed and a caching mechanism can gracefully deal with user requests when recovering.

Still, the idea of ROC that applying software fault injection to infer the fault path and dependency map also gives us inspire to discover service dependency. Fault injection \[17\] is widely used to evaluate the reliability of computer systems. Researchers and engineers have created many novel methods to inject faults, which can be implemented in both hardware and software. As to the software implemented fault injection (SWIFI), Jaca \[9\] is a good fault injection pattern system based on Java reflection and AOP. It gives us many hints and experiences to implement the automated evaluation of correlated failures via fault injection. But it cannot deal with the problem discussed in this paper because Jaca does not consider the semantics of J2EE services.

The contributions of this paper can be summarized in two aspects. First, we
identify a special kind of failure, called correlated failures, which are caused by the interdependent relationship among middleware services. This kind of failure impacts the system reliability significantly and must be considered seriously. Then we propose a framework to recover the correlated failures. The framework covers the discovering, detecting and recovering the correlated failures, represents a coordinated recovery model, and also gives out some supporting mechanisms and techniques. However, there are still some open issues to be addressed in the future.

The evaluation of correlated failures as well as the discovery of interdependent relationships among middleware services is an important but hard part of improving system reliability using coordinated recovery. As discussed in section 4.2.2, reviewing system documentations and implementation can help to find out the correlated failures, but it has some difficulties and limitations. Our solution, JARME, is easy to perform and has a good effect. But there are also some limitations. First, the effectiveness of JARME largely depends on the coverage of test cases. A set of test cases with broader coverage helps JARME discover more correlated failures. If test cases used in JARME do not cover all scenarios, there may be correlated failures which are not triggered in the test but will occur in the real world. Second, the services can interact with each other not only by method call but also by shared memory or global variables. JARME can only discover the correlated failures caused by method call. Third, JARME can only detect faults that are manifested as exceptions. Those faults that are manifested as others cannot be discovered. Furthermore, history data of application failures should be used to avoid their reoccurrence.

In the experiments, we cannot recover all transactions and there are still nine failed transactions in Case I. At the time the data service fails, there are some operations being performed and exceptions are thrown. We do want these exceptions to be thrown through the invocation chain because there may be other services which want to know the failure of the data service. We leave these transactions un-recovered to inform other services. A possible solution to both recover all failed transactions and inform all services is to throw fake exceptions. But there are still some technical difficulties, such as the definition of fake exceptions and the system’s consistency, are not resolved at present. But we still can recover all failed requests in some certain situation, as in Case III. If the correlated failures are detected in the depending service only, there will be no need to re-throw the caught exceptions. With the restriction that the requests are idempotent which means the requests get the same responses and the reduplicate executions do no harm to the entire system, the failed requests can be redone so that all failed requests can be recovered.

Another weakness of current work is the framework considers only two correlated failures in a coordinated recovery process. A more complex dependent relationship between three or more services needs more complex coordinated recovery. The approach to dealing with two dependent services above should be extended to deal with more dependent services. Moreover, the framework does not consider the concurrent failures in middleware services as some work has done \cite{37}. We believe above two problems are also influential in reliability and availability of middleware in a realistic scenario and will be addressed in our future work.

We have not talked about the fault tolerance of recovery facilities especially the Recovery Coordinator by now. Indeed if faults occur in the Recovery Coordinator,
the whole recovery framework cannot work any more. It is a single point of failure. Currently we implement the fault tolerant Recovery Coordinator by restarting when faults are detected because it can be regarded as a service either. But we have not taken into account the situation when faults occur in the Recovery Coordinator during performing some recovery processes. In our point of view, a replica mechanism for Recovery Coordinator may solve the problem because the shortcomings of replica-based approach can be avoided in such a small (but important) component.

7 Conclusion

The correlated failures of middleware services are threats to the dependability of middleware-based systems. Considering the interdependent relationships, both the recovery performance and recovery efficiency can be improved. This paper analyzes the interdependent relationships which cause the mutually dependent faults, gives classifications to the interdependent relationships, mutually dependent faults and correlated failures, and proposes a general framework for discovering, detecting and coordinated recovering correlated failures. The most important rationale of our work is that the detecting and recovering of correlated failures should be under central coordination. The coordinated recovery model helps to partition and decouple the detecting and recovering modules. The using of SWIFI technique reduces the workload and difficulties of discovering and defining correlated failures. Features required making service recoverable along with some sample implementations guides the users to add recoverable ability to standard middleware services. Several experiments are also represented and each of them shows different benefits of the coordinated recovery. Our supporting framework is general enough for J2EE and has been demonstrated on three J2EE middleware products.

Acknowledgements

This effort is sponsored by the National Basic Research Program (973) of China under Grant No.2002CB312003; the National Natural Science Foundation of China under Grant Nos.90612011, 90412011, 60403030; and the IBM University Joint Study Program The authors would like to thank the editors and reviewers for their constructive comments which greatly improved this paper.

References


