Towards Self-Healing Systems via Dependable Architecture and Reflective Middleware

Hong MEI\(^1\), Gang HUANG\(^1\), Wei-Tek Tsai\(^2\)

\(^1\)School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, China. E-mail: meih@pku.edu.cn, huanggang@sei.pku.edu.cn
\(^2\)Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287-8809 wtsai@asu.edu

Abstract

Self-healing systems focus on how to reducing the complexity and cost of the management of dependability policies and mechanisms without human intervention. This position paper proposes a systematic approach to self-healing systems via dependable architecture and reflective middleware. Firstly, the differences between self-healing systems and traditional dependable systems are illustrated via the construction of a dependable computing model. Secondly, reflective middleware is incorporated into the dependable computing model for investigating the feasibility and benefits of implementing self-healing systems by reflective middleware. Thirdly, dependable architectures are introduced for providing application specific knowledge related to self-healing. Fourthly, an architecture based deployment tool is implemented for deploying dependable architectures into heterogeneous and distribute environments. Finally, an architecture based reflective J2EE application server, called PKUAS, is implemented for interpreting and enforcing dependable architectures at runtime, that is, discovering or predicting and recovering or preventing failures automatically under the guide of dependable architectures.

1. Introduction

In terms of the concepts and taxonomy built up by Avizienis et al. in [2], “the dependability of a system is the ability to avoid service failures that are more frequent or more severe that is acceptable”. 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, as shown in Figure 1.

![Figure 1. Dependable Computing Model](image)

The events occurring in the system will be collected in an active or passive manner for discovering the threats to the dependability, including the faults, errors and failures. Whether the monitored events indicate some threats or not is measured by the attributes of dependability, including the availability, reliability, safety, integrity and maintainability. For example, assume the requirement of the reliability is that “if the system cannot deliver correct services more than 2 times per 1000 requests, the system is not reliable”. If the incorrect services occur at the No. 1, 2, 1001 and 1002 requests, only the last one indicates a system failure. Once a failure is discovered or predicted, the system will try to deal with the failure by means of fault prevention, fault tolerance, fault removal and fault forecasting. No matter which means is adopted, the system is usually changed more or less. After that, the system may evaluate the effects in terms of the attributes of the dependability. If the failure can not be recovered or prevented, the system may have to try other means.

There are complex relationships among different attributes, different failures, different means and different adaptations. For example, firstly, an event may indicate more than one failure when considering multiple attributes at the same time. Then, the event may activate multiple
recovery processes and concurrent adaptations, which may bring unexpected results. On the other hand, a trade-off among multiple attributes can take place for determining which failure is indicated by the event. Secondly, a failure may be indicated by more than one event. When two isolated events occur, it is very difficult to determine whether two failures or only one occur. Thirdly, a failure may cause another failure. When multiple failures are discovered in a short time, there may have a root-cause failure that causes all other failures. Then only the root-cause failure requires to be recovered or it is recovered at first. Other samples include the trade-off for the conflicts among the attributes, the change impact analysis of the adaptations, and so on.

To avoid or simplify the above complex relationships, a dependable system often deals with several independent scenarios, which involve a limited number of predictable or controllable attributes, failures, means, and adaptations. Such a way works well for the relatively static and closed environments. However, the extremely open and dynamic natures of today's Internet and its applications make the above elements related to the dependability flourishing, unpredictable and uncontrollable. It implies that the complex relationships among the elements cannot be avoided or simplified by isolated scenarios any more. In other words, all attributes, threats, means, and adaptations available in a dependable technology should be managed in a uniform and systematic way, that is, all attributes and their relationships have to be clearly specified and, if necessary, traded off; all threats have to be monitored and measured as a whole and the root-cause failure should be discovered; all means have to be evaluated and selected in terms of the contexts; change impacts of all adaptations have to be analyzed and the adaptations may be performed in a collaborative manner.

Obviously, the management of the above relationships is very complex, challenging, time-consuming, and error-prone. Recently, autonomic computing is becoming a hot topic in computer technologies because it reveals the practical and urgent requirements on the computer-based system that can manage itself and then promise customers to drastically decrease the cost of management [11]. One of the four major characteristics of an autonomic computing system is to free people to discover, recover, or prevent system failures, which is called self-healing. Distinguished with traditional dependable policies and mechanisms, self-healing focuses on how to identify, organize, reason, and control the complex relationships among the dependability elements. In that sense, self-healing is a natural way to make systems dependable in the open and dynamic Internet.

Technically, a self-healing system can be divided into two parts performing different computations: basic computation utilizes computer and network resources to solve problems in its application domain; self-healing computation, distinguishing self-healing systems with other computational systems, is responsible for making basic computation dependable via four steps: firstly, monitoring and collecting the states and behaviors of the basic computation; secondly, measuring and analyzing the collected data to discover or predict the failures; thirdly, planning and deciding when and how to recover or prevent the failures; fourthly, adjusting the states and behaviors of the basic computation to perform the self-healing plan. In fact, the four steps exist in all dependable systems, which can be partially proved by the fact that the steps can be concluded from the dependable computing model shown in Figure 1. For traditional dependable systems, the four steps are restricted for one or few given scenarios due to all related elements and details are predictable and controllable. For self-healing systems, the steps can deal with much more scenarios due to part of the elements and details can be unpredictable and uncontrollable.

Definitely, it is very challenging to achieve self-healing attribute. This position paper discusses how to make a system self-healing via architecture based reflective middleware. Firstly, we will incorporate the reflective middleware into the dependable computing model to provide self-healing systems with a sophisticated framework for monitoring and adjusting runtime systems. Then, we will introduce software architectures to organize and represent the management information related to self-healing. Finally, we will demonstrate the approach with an empirical study of PKUAS, an architecture based reflective application server compliant with J2EE (Java 2 Platform Enterprise Edition).

The rest of the paper is organized as follows. Section 2 discusses the benefits of enabling self-healing systems via reflective middleware; Section 3 describes the introduction of dependable architectures into self-healing; Section 4 identifies the challenges to the deployment of dependable architectures and implements an architecture based deployment tool; Section 5 briefly introduces PKUAS for interpreting and enforcing dependable architectures; Section 6 concludes the paper and identifies the future work.

2. Reflective Middleware for Self-Healing

2.1. Dependable Systems Enabled by Middleware

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 2. 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 is 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, retail, etc. Thirdly, middleware provides some facilities, such as component models, to help the development and deployment of distributed systems.

![Figure 2. Self-healing by the Application and Middleware Respectively.](image)

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. Recall the dependable computing model, 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 self-healing, the proliferation means that middleware has to take much more responsibilities for self-healing than the applications and underlying operating platforms.

For the middleware based systems, self-healing can be performed by the application, the middleware and the underlying operating system respectively, as shown in Figure 2. For the application, it can detect the faults, errors and failures taking place at the application level and recover part of them due to due to 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 at the application level and middleware level due to it has no or little knowledge of the semantics of the whole system. For middleware, only the faults and errors that are embedded in the application code and the implementation of the underlying operating system and do not fire any alarms, like exceptions, cannot be detected and recovered due to the middleware controls the execution of the application via utilizing the underlying operating system under the guide of the knowledge of the whole system.

In fact, some special failures can only be discovered and recovered by the three layers respectively. Thus, it is an ideal way to perform self-healing at all levels. However, since middleware achieves the best-of-the-breed tradeoff between the control and knowledge, we think middleware should play a centric role in self-healing systems.

Technically, to support self-healing, middleware should be able to monitor (observe runtime states and behaviors), reason (analyze the states and behaviors and then decide when and what to change), and control (perform changes of runtime states and behaviors) the whole middleware platform and part of middleware applications. For a given middleware, its observable and changeable states and behaviors at runtime are determined by the middleware itself, while the analysis and decision making depend on not only the middleware but also the deployed applications and the responsibilities of the administrators. For example, two applications use a special component. The response time of a given method exceeding 3 seconds may be considered as a failure in one application while as a success in another application. No matter what applications are using the component, the middleware can detect the response time of the given method. However, for discovering the failure, the middleware must have enough knowledge of the applications. Thus, middleware have to provide a sophisticated framework specific to middleware for monitoring and controlling itself while provide a general-enough framework to reason itself depending on the middleware, applications and administrators. We find that reflective middleware has the framework for monitoring and controlling.

### 2.2. Reflective Middleware

Reflection, also known as computational reflection, is originated by B.C. Smith to access and manipulate the LISP program as a set of data in execution [22]. Since it helps to achieve flexible and adaptive systems, reflection is propagated into operating system, distributed system, and middleware one by one. In the past several years, many reflective middleware are developed as an extension to the common middleware. DynamicTAO [12], OpenCORBA [13] and FlexiNET [7] are reflective

Figure 3 illustrates the basic concepts of reflection. A reflective system is a computational system with two levels. The base level consists of base entities that perform the usual functionality of the system, that is, the basic capability of a computational system regardless of whether it is reflective or not. In details, it builds a model to represent the problem domain and then reasons and manipulates on the model to solve the problems. The meta level consists of meta entities that perform reflection on the system. It builds a model to represent the base level.

2.3. Self-Healing Systems Enabled by Reflective Middleware

From the above analysis of dependable systems and reflective middleware, it can be concluded that dependable systems can utilize reflective middleware for self-healing as shown in Figure 4: firstly, reflective computation collects the states and behaviors of basic computation; secondly, self-healing computation measures and analyzes these data and decides when and what to change; finally, reflective computation performs the changes on basic computation.

2.3. Self-Healing Systems Enabled by Reflective Middleware

From the above analysis of dependable systems and reflective middleware, it can be concluded that dependable systems can utilize reflective middleware for self-healing as shown in Figure 4: firstly, reflective computation collects the states and behaviors of basic computation; secondly, self-healing computation measures and analyzes these data and decides when and what to change; finally, reflective computation performs the changes on basic computation.
Though reliability is often considered as a typical sample in the quality related analysis and reasoning on software architecture, dependability is only just emerging as an important issue in software architecture recently, which can be witnessed by three workshops on architecting dependable systems (WADS) from 2002 [14][15][16]. Current work on software architectures for dependable systems can be divided into two categories. On one hand, the specific dependable systems, including fault-tolerant middleware, are studied thoroughly so as to identify, abstract, and classify the characteristics of dependability at architectural level. On the other hand, many methods of architecting dependable systems are proposed with some simple case studies.

Current architectures of dependable systems do not take middleware into consideration. They become much more complex when dependability computing is required. For example, replication of servers is a typical and relatively simple technique to avoid the failures caused by the absence of single server in a client-server system. The left side of Figure 5 shows the possible architectures of the client-server system using replication of servers.

[Diagram showing the architectures]

Figure 5. A Sample of Dependable Architectures Aware of Middleware vs. Unaware of Middleware

If the dependability requirement is that a client cannot endure the loss of any work in progress when the connected server fails and the failed request must be automatically re-directed to another available server and get the correct result, the architect can design the passive replication of the server, which is shown in the top-left of Figure 5. In the active replication, the state of request processing is actively and completely shared among all replicas. Then, the proxy has to dispatch a request to all replicas that have to synchronize their states. Unless all replicas fail, the proxy can receive at least one correct result.

In practice, the switch among active and passive replicas, the instantiation of passive replicas, the dispatching to all replicas, and the synchronization of the states of all replicas require much more detailed description to specify their semantics completely and precisely. If many components require the replication, the architecture will become inevitably large and complex. Moreover, the replication is a usual and relatively simple means to dependability. If the system requires other complex means, the dependable architecture may become too large and complex to understand and reason.

3.2. Modeling Dependable Architectures Enabled by Reflective Middleware

In our point of view, architecting dependable systems will produce some design decisions on how to discover and recover predictable failures. If the middleware cannot provide supporting mechanisms, the design decisions will motivate complex design and have to be implemented by hand-writing code. On the other hand, if the middleware, e.g., reflective middleware, can provide plentiful supporting mechanisms and such capabilities are considered by the architects, the design decisions related to dependability could be directly implemented by middleware and the dependable architecture will be simplified significantly.

Let’s take the replication of servers as example again. The three types of replication are directly supported by middleware. It means that the architect can only specify which components should be replicated as which type. The right side of Figure 5 shows the simplified architectures of the client-server system using replication...
provided by middleware. The stereotypes of the component, including Passive-Cold-Replica, Passive-Hot-Replica and Active-Replica, can be correctly, precisely and automatically interpreted by middleware. So, architecting dependable systems with awareness of middleware can not only simplify the architecture but also provide plentiful knowledge for middleware to heal runtime systems automatically.

The modeling of dependable architectures enabled by middleware requires special notations or existing notations with some extensions. In our experiences, the stereotypes, tagged values, templates, aspects, connectors, styles and patterns can represent lots of dependability capabilities of middleware. However, it is better to standardize the notations for middleware so that the dependable architectures designed by any modeling tools can be understood and interpreted by any middleware.

4. Deployment of Dependable Architectures

Before the whole system executes, the application has to be deployed into the middleware platform. When deploying a dependable system, the design decisions related to dependability should be transferred into middleware, which then can parse the knowledge and guide the self-healing actions. Particularly, some design decisions related to dependability have to be dealt with in the deployment.

The major work in the deployment is to provide lots of information for configuring middleware so that the middleware can execute the application correctly and efficiently. For example, the deployment of a J2EE application has to produce an XML file, called deployment descriptor. There are more than 25 items for an EJB, more than 30 items for a Web Application and more than 20 items for a J2EE Applications, including the names of EJBs, the declaration of required resources, security realm and roles, component names for runtime binding, and so on [24].

Consider the above client-server example again. Since the replication is usually supported by the clustering mechanism, the replicated server component has to be deployed into a cluster. If the architect specifies the minimal number of replicas, the deployer has to select such a cluster that contains machines equal to or more than the replicas. If the clients can come from anywhere in Internet, the deployer has to select such a cluster that has Internet connections. If the architect specifies that the cost of the replication should be reduced as much as possible, the deployer may have to select the cheapest cluster.

To help the deployer to understand and keep the consistency with the design decisions, it is necessary to do all work related to the deployment from the perspective of software architecture. Figure 6 shows an architecture based deployment tool for J2EE applications, called CADTool [10]. The “deploy” panel shows the software architecture of Java Pet Store (JPS). JPS is one of the sample applications for J2EE Blueprints, demonstrating how to use the capabilities of the J2EE platform to develop flexible, scalable, cross-platform e-business applications.

Figure 6. Architecture based Deployment Tool

CADTool can facilitate the deployment with four features. Firstly, it can visualize architecture models in the development. In current version, CADTool reuses the graphical elements of ABCTool, which supports architecture modeling with ABC/ADL in a visual way [19]. If the deployable package contains the ABC/ADL architecture description, CADTool can display the syntax and semantics information produced in the development. If the deployable package contains the layout description of the architecture, CADTool can display the architecture in the same layout as that in the development, which helps to understand the intention of the designers. If the deployable package does not contain the two descriptions, CADTool can automatically construct the architecture from the individual deployable components. However, the last case is not desired because the architecture lacks enough information derived from the development.

Secondly, CADTool can visualize the servers and their capabilities. Using reflective mechanisms of PKUAS, CADTool can automatically collect and display the servers’ information, such as CPU utilization, memory utilization, throughout, etc. The information is useful to determine which components should be deployed into which servers. They also help to investigate whether the deployment works well. For example, the CatalogEJB consumes much CPU time. If the component is deployed into the Server1, the CPU utilization of the Server1 may exceed 90% and the Server1 becomes unstable and easy to crash. Then, it’d better un-deploy the CatalogEJB in the Server1 and re-deploy it into the cluster.
Thirdly, CADTool supports the deployment of components in a drag-and-drop manner. With the above two visual elements, a component can be easily deployed into a server just through dragging the component and dropping it on the target server or vice versa. In traditional deployment tools, the deployer has to connect to a given server, load the components to be deployed into the server, and repeat the work again for another server. If a component is deployed into a server, the box representing the component will have a shadow with the same color as the server. In the Figure 6, the red, blue and green colors identify Server1, Server2 and Cluster respectively.

Fourthly and finally, CADTool can automatically calculate some system-level or scenario-level properties. There are many successful case studies on the quantitative and qualitative evaluations of the given architecture models. However, some properties may be wrongly predicted in the design phase and should be re-evaluated in the deployment. Specially, some properties may be only available after the system running for a period, such as the response time and throughput. That means the deployment may not meet the requirements related to these properties. Then the whole or part of system has to be re-deployed with the actual properties. Currently, CADTool can automatically calculate the response time, throughput and reliability of a given use case.

5. Enforcement of Dependable Architectures

After the dependable architectures are deployed, they must be interpreted and enforced by reflective middleware at runtime so as to implement the design decisions related to dependability. However, current reflective middleware represents reflective operations as a set of fragmented and irresolute items, which are necessary but insufficient to understand and reason the semantics of reflective operations and their relationships. Such representation is very different with dependable architectures. As a result, reflective middleware may not be able to correctly and precisely perform the changes desired by the dependable architectures. That means the changes made on the runtime system may be less or more than the desired changes. To keep the middleware having the same perspective on the whole system as dependable architectures, we implement an architecture based reflective middleware, called PKUAS [9][18], which supports the monitoring and controlling of runtime systems from the perspective of software architecture.

As shown in Figure 7, the states and behaviors of the platform and applications of middleware can be observed and adapted from the platform architecture and the application architecture respectively. The platform architecture represents the implementation of the platform of middleware as a set of components and connectors (called platform components and platform connectors). The applications of middleware are invisible or only represented as the attributes of some platform components. For example, J2EE application server consists of containers and services and the J2EE application consists of EJBs or Servlets. In the platform architecture, the containers and services are represented as platform components; their interactions or dependencies are represented as platform connectors; the EJBs or Servlets are represented as the attributes of the containers.

![Figure 7. Architecture-based Reflective Framework](image-url)

On the other hand, the application architecture represents the application of middleware as a set of components and connectors (called application components and application connectors). The functions of the platform of middleware are typically represented as the constraints or attributes of application components and application connectors. For example, J2EE security and transaction services are represented as the security and transaction constraints on the EJBs or Servlets.

The reflective framework is the kernel that manages the redundant resources to discover and recover faults, errors and failures. The question is how can we make the reflective framework itself dependable? The question could be answered by applying the reconfigurable Service-Oriented Architecture (SOA) [25] and the seamless combination of SOA and reflective middleware. Using the SOA, all components in the reflective framework are replaceable at runtime without stopping the operation. Another purpose of SOA is to dynamically update the system when system environment and the requirements are changed.

6. Conclusion and Future Work

Self-healing systems free people to manage the diverse and complex policies and mechanisms related to dependability. However, every activity of self-healing is very challenging. Based on previous work on software architecture and reflective middleware, this position paper proposes to implement self-healing systems via
dependable architecture and reflective middleware in a systematic way. In this approach, dependable architectures provide application specific knowledge related to self-healing while reflective middleware not only provides a sophisticated framework for monitoring and controlling the runtime system but also interprets and enforces dependable architectures. In other words, reflective middleware can automatically discover or predict and recover or prevent system failures under the guise of dependable architectures. This approach is empirically demonstrated by an architecture based deployment tool, called CADTool, and an architecture based reflective middleware, called PKUAS.

There are still many open issues remaining to achieve self-healing middleware, including the methodology to model dependable architectures with existing architecture description languages and visualized tools, the model of self-healing middleware describing the attributes, threats, means and adaptations related to dependability in an understandable and easy-to-use manner, the detection of root cause in multiple correlated and simultaneous failures, the trade-off among different dependable attributes, the change impact analysis of adaptations, the automatic deployment and re-deployment of dependable architectures, and so on. Moreover, many more self-healing cases and real systems should be studied and evaluated.

Acknowledgements

This effort is sponsored by the National Key Basic Research and Development Program (973) under Grant No. 2002CB31200003; the National High-Tech Research and Development Plan of China (863) under Grant No. 2004AA112070; the National Natural Science and Development Plan of China (863) under Grant No. 2002CB31200003; the National High-Tech Research and Development Program (973) under Grant No. 2002CB31200003; the National High-Tech Research and Development Program (973) under Grant No. 2002CB31200003; the National High-Tech Research and Development Program (973) under Grant No. 2002CB31200003.

References