PKUAS: An architecture-based reflective component operating platform

Conference Paper · June 2004

DOI: 10.1109/FTDCS.2004.1316609 · Source: IEEE Xplore

2 authors:

Hong Mei
Xi'an Jiaotong University
309 PUBLICATIONS **4,957** CITATIONS

SEE PROFILE

Gang Huang
Southeast University (China)
227 PUBLICATIONS **2,135** CITATIONS

SEE PROFILE

PKUAS: An Architecture-based Reflective Component Operating Platform

Hong MEI, Gang HUANG
School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, China.
E-mail: meih@pku.edu.cn, huanggang@sei.pku.edu.cn

Abstract

Reflective middleware is the major approach to improving the adaptability of middleware and its applications. Current researches and practices pay little attention on the usability of reflective middleware. There also lacks a systematic way to adapt runtime system via reflective middleware. This paper presents the design and implementation of PKUAS, an architecture-based reflective component operating platform compliant with Java 2 Platform Enterprise Edition. PKUAS constructs and represents its platform and applications from the perspective of software architecture so as to provide an understandable, user-friendly and systematic way to use reflective middleware.

1. Introduction

Nowadays, Internet becomes a new dominant runtime environment of software with extremely open and dynamic characteristics. As a result, middleware, as the popular operating platform over Internet, has to be much more adaptable than ever. Traditional middleware has the philosophy of "black-box" reuse, which hides the heterogeneity of underlying operating systems, networks and programming languages. Though the "black-box" reuse frees the developers from dealing with the heterogeneity, it also prevents the maintainers from adapting the middleware for changing environments or requirements. Then, traditional middleware cannot achieve the high adaptability required by extremely open and dynamic Internet.

Being one of the hot topics in the researches and practices on next generation middleware, reflective middleware is considered as the fundamental approach to adaptable middleware [1]. Compared to traditional middleware, reflective middleware makes the runtime states and behaviors internal of middleware platform and applications observable and adaptable. In other words, reflective middleware employs the philosophy of "grey-box" reuse to become much more adaptable.

Though reflective middleware receives much more attentions from the academic and industrial communities recently, many challenges remain to be addressed. Firstly, the usability of reflection is still poor. Most reflective middleware represent their states and behaviors as a set of fragmented and irrelative items. Such representations are necessary but insufficient to understand and reason middleware platform and applications. Secondly, it is necessary to systematically adapt runtime systems via reflective middleware because of the consensus that the adaptability of software systems should be considered in the whole lifecycle. But current researches and practices focus on implementing reflective mechanisms, that is, how to adapt, but pay little attention on why, what and when to adapt. Thirdly and finally, to the best of our knowledge, there seems no work on systematically introduce reflection into J2EE (Java 2 Platform Enterprise Edition), which is one of the most popular middleware.

This paper presents an architecture-based component operating platform, called PKUAS (PeKing University Application Server) and compliant with J2EE, to address the above issues. The rest of this paper is organized as follows: Section 2 introduces some work related to reflective middleware; Section 3 discusses some basic ideas of architecture-based reflective framework; Section 4 details the design and implementation of PKUAS and evaluates its performance. Section 5 summarizes the contributions and identifies the future work of this paper.

2. Related Work

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 [16]. Figure 1 illustrates the fundamental concepts of reflection. A reflective system is a computational system having two levels. The base level consists of base entities that perform the usual functionality of the system, that is, the basic ability 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.

Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'04)
0-7695-2118-5/04 $20.00 © 2004 **IEEE**

This model, called self-representation of the system, is causally connected with base entities, that is, changes of base entities will immediately lead to corresponding changes in self-representation, and vice versa [12]. The computation in the meta level is to guarantee the causal connection between self-representation and base entities. Then, a reflective system can be formally defined as the computational system having the ability, called reflection, that its internal states and behaviors can be accessed and modified through its causal connected self-representation.

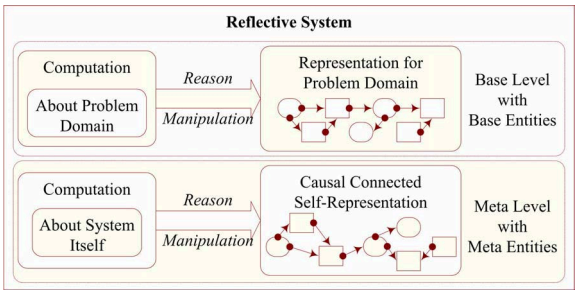

Figure 1. Architecture of Reflective System

In the past several years, many reflective middleware are developed as an extension to the common middleware. DynamicTAO [10], OpenCORBA [11] and FlexiNET [7] are reflective CORBA (Common Object Request Broker Architecture) platforms. mChaRM [3] is a reflective RMI (Remote Method Invocation) platform. OpenORB [2] adds reflection ability into COM (Component Object Model). All of these middleware have the limitations of usability and reflective capability more or less and do little efforts on the systematic adaptation via reflective middleware.

3. Overview of PKUAS

Considering the challenges to reflective middleware, it could be concluded that the self-representation should represent the states and behaviors of middleware platform and applications in a uniform and understandable way. Such self-representation should also facilitate to identify and analyze changes enabled by reflective middleware in the whole software lifecycle. Since software architecture (SA) helps to understand large-scale software systems and plays an important role in software development [5], it is a natural idea to make SA act as the self-representation of reflective middleware.

SA describes the gross structure of a software system with a collection of components, connectors and constraints [15]. In general, SA acts as a bridge between requirements and implementation and provides a blueprint for system construction and composition. It helps to understand large systems, support reuse at both component and architecture level, indicate the major components to be developed and their relationships and constraints, expose changeability of the system, verify and validate the target system at a high level and so on [5]. Since most common middleware provide supports for component based development, like CORBA Component Model and Enterprise JavaBean, and even themselves are constructed from components, their platform and applications can be sufficiently and suitably represented by SA.

3.1. Architecture based Reflective Framework

Compared to SA in development, SA introduced into reflective middleware is available at runtime and provides a more concrete view of the runtime system with much more information. We call such SA as runtime software architecture (RSA) [9]. Figure 2 shows the framework of such architecture-based reflective middleware.

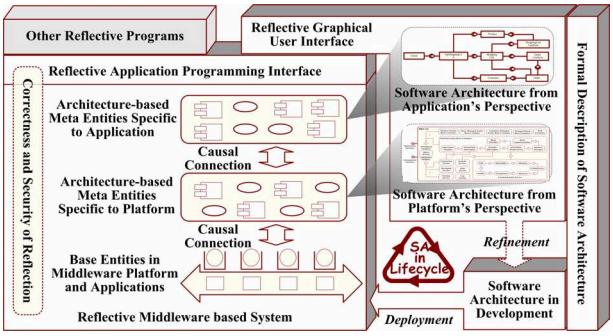

Figure 2. Architecture-based Reflective Framework

The states and behaviors of middleware platform and applications can be observed and adapted from the perspectives of the platform RSA and application RSA respectively. The platform RSA represents the implementation of middleware platform as components and connectors. Middleware applications are invisible or represented as the attributes of some components. For example, J2EE application server consists of containers and services and the J2EE application consists of EJBs or Servlets. In the platform RSA, the containers and services are represented as components; their interactions or dependencies are represented as connectors; and the EJBs or Servlets are represented as the attributes of the containers. On the other hand, the application RSA represents middleware application as components and connectors. Middleware platform are typically represented as constraints or attributes of components and connectors. For example, J2EE security and transaction services are represented as the security and transaction constraints on the EJBs or Servlets.

The platform RSA precisely represents the actual

implementation of middleware platform, while the application RSA provides a much more understandable view for the states and behaviors of runtime systems. Specially, if SA description in software development is available, the application RSA can enrich its semantics with plentiful design information. To ease the work of analyzing the design artifact and verify the correctness of reflection, RSA should be formally described by an extended architecture description language. Such formal description also helps to keep consistency and traceability of SA models between software development and runtime, which is necessary to achieve the systematic adaptation based on reflective middleware.

3.2. Process Model

Introducing SA into reflective middleware not only improves the usability of reflection, but also makes SA explicitly available in the whole software lifecycle. As a result, it is feasible to systematically adapt middleware platform and applications via reflection in an architecture based way. More details will be discussed with an architecture based process model, as shown in Figure 3.

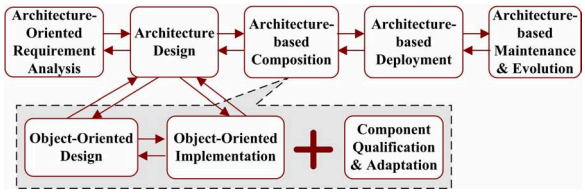

Figure 3. ABC Process Model

ABC (Architecture based Component Composition) is a software reuse methodology that supports to build a software system with pre-fabricated components under the guide of SA [13]. In ABC, SA acts as a blueprint in the whole software lifecycle. In requirements specification, the concepts and principles of SA are introduced into this phase to achieve the traceability and consistency between requirement specifications and system design. In this phase, there is no actual SA but only the requirement specifications of the system to be developed, which are structured in the way similar to SA. In architecture design, requirements specifications are refined and some overall design decisions are made. After architecture design, the components, connectors and constraints in the reusable assets repository will be selected, qualified and adapted to implement the target system. However, there are still some elements unable to be implemented by reusable assets. These elements have to be implemented by hand in object-oriented languages or other ones. Before the implementation of the system being executed, it must be deployed into middleware platform. In this phase, SA should be complemented with some information so that middleware can install and execute the system correctly. Typically, the information includes declaration of required resources, security realm and roles, component names for runtime binding, and so on.

In some sense, the development of a system in ABC can be considered as a series of automated refinement and transformation of SA models. The syntax and semantics of SA would become more accurate or complete after every refinement or transformation. From the view of software lifecycle, the observation and manipulation via reflective middleware can be considered as the activities in software maintenance and evolution. Then, RSA has the most accurate and complete details describing the final system. The adaptation via reflection can be verified and validated with plentiful information aggregated from requirements phase to runtime. At the same time, some changes may be predicted in system development and deployment. Such information can guide the adaptation via reflection. Consequently, the adaptation via reflection could be performed in a systematic way.

4. Implementation

4.1. Componentized Structure

PKUAS is a J2EE-compliant application server which is the platform including J2SE, common services and one or both of Web Container and EJB Container [18]. It provides all functionalities required by J2EE v1.3 [18] and EJB v2.0 [17] in its componentized structure, as shown in Figure 4.

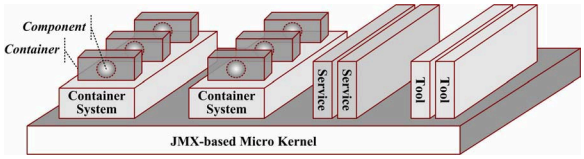

Figure 4. Componentized Structure in PKUAS

- Container system and container: a container provides a runtime space for the components in the deployed applications with lifecycle management and contract enforcement [18]. PKUAS implements standard EJB containers for stateless session bean, stateful session bean, bean-managed entity bean, container-managed entity bean and message-driven bean [17]. One instance of a container holds all instances of one EJB. And a container system consists of the instances of the containers holding all EJBs in an application. Such organization of the containers facilitates the configuration and management specific to individual applications, such as security realm per application and architectural information of the application.
- Service: it provides the common functions, like

naming, communication, security, transaction and log. The naming and communication services provide an interoperability framework that enables the components deployed in PKUAS to interact with each other and other components outside PKUAS through multiple interoperability protocols [8].

- Tool: it provides functions to facilitate the operation of PKUAS, such as deployment and management.
- Micro kernel: it provides a registry and an invocation framework for the above platform components and other management entities, like class loading, relation, timer and monitor. In fact, it is implemented as a JMX MBeanServer. JMX (Java Management Extensions) is a specification that defines the architecture, design patterns, APIs and services for application and network management and monitoring in Java programming language [20]. In JMX, both managing components and managed components are implemented as MBeans, which support plug-and-play dynamically. The MBeanServer is a registry for MBeans and provides an invocation framework for MBeans in the same Java Virtual Machine (JVM). Being programmed in Java, PKUAS is executed in JVM.

Such componentized structure is essential to reflective middleware because it can help to identify entities to be reflected clearly and manipulate a given entity independently from the others [4]. Considering reflection, the container systems, containers, components and services are wrapped by meta-objects and plugged into the micro kernel. Then the states and behaviors of these platform components can be observed and changed through their reflective wrappers.

4.2. Class Loader Hierarchy

In Java, the class loaders are responsible for loading class definitions from the ".class" files [6]. The default class loader in JVM only loads classes from the directories or ".jar" files specified by the "java.class.path" property of JVM and does not reload the class definition with the same name as an already loaded class. Obviously, it cannot support the replacement of classes required by reflection. Then PKUAS implements a set of class loaders that constitute a hierarchy as shown in Figure 5.

Being the top of the hierarchy, System Class Loader is responsible for loading classes of the implementation of PKUAS, which are visible to all classes in PKUAS and cannot be replaced at runtime. The second layer consists of three types of "brother" class loaders. Service Class Loader is responsible for loading classes specific to one service, e.g., security service, transaction service and communication service; EJB Application Class Loader is responsible for loading classes from the ".ear" or ".jar" file that contains EJB implementations; and Web Application Class Loader is responsible for loading classes from the ".ear" or ".war" file that contains JSP/Servlet and other web pages. The "brother" relationship ensure the services, EJB applications and Web applications cannot access each other without the help of PKUAS micro kernel. Then the addition of services, EJB applications and Web applications can be achieved by adding new class loaders to load the specified classes. And the replacement can be achieved by removing the old class loaders and adding new class loaders sequentially. EJB Application Class Loader will create an EJB Class Loaders per EJBs, and EJB Class Loader will create three class loaders to load the contract, implementation and constraints respectively. Then the whole of an EJB or its contract, implementation and constraints can be added or replaced independently by adding and removing the corresponding class loaders.

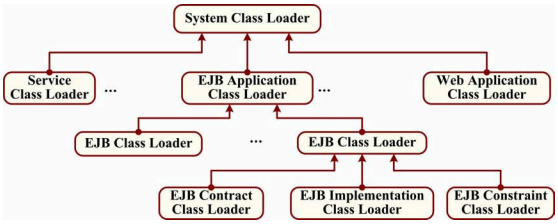

Figure 5. PKUAS Class Loader Hierarchy

4.3. Reflective Container

In PKUAS, one container instance provides a runtime space for all instances of a component and holds all details of the component. In that sense, to observe and manipulate application components in the application RSA is equal to reflect their containers in the platform RSA. The observation can be easily implemented by exposing internal attributes of the container and application component. The addition and removal of the whole component can be easily implemented by creating or releasing the container instance and publish or withdraw its remote reference in the naming service. The replacement of the whole component can be easily achieved by replacing its container. The replacement of the component interface can be achieved by unloading the old interfaces and loading new ones. Component constraints are enforced by a set of interceptors, which can be dynamically added, removed and replaced.

Considering the lifecycle of component instances, reflection about component implementation becomes much more complex. Typically, an EJB instance has five states in its lifecycle, including Loaded, Instantiated, Sessional, Transactional and Serving. The Loaded state indicates that the implementation classes of an EJB are

loaded into PKUAS after its deployment and unloaded after its un-deployment. After the EJB is invoked at first time, an instance of the implementation is created and stays in the Instantiated state if it is not associated with any client. The Sessional instance means that the instance is associated with a client and cannot deal with the invocations from other clients. If an instance is in an uncompleted transaction, it becomes Transactional and has to keep the transactional information spanning multiple invocations. The Serving instance means that the instance is dealing with an invocation by the implementation classes. Table 1 summaries the details of replacing EJB instances in terms of the five states.

Table 1. Replacement of EJB instances

EJB State	Replacement Details
Loaded	(1): Load new classes
Instantiated	(2): Release old instances + (1) + create new instances
Sessional	(3): (2) + restore attributes
Transactional	(4): (3) + restore transaction context
Serving	Not supported

4.4. Reflective Interoperability Framework

PKUAS defines a reflective interoperability framework to support the components deployed in PKUAS to interact with the components deployed in other component frameworks, as shown in Figure 6.

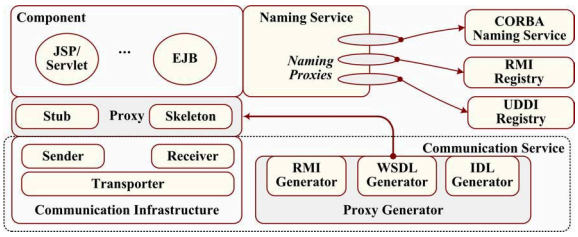

Figure 6. PKUAS interoperability framework

Communication infrastructure provides the necessary functions required by particular interoperability protocols. Referring the concepts and design patterns of CORBA Extensible Transport Framework [14], PKUAS can plug-play not only multiple transport protocols, e.g., TCP/IP, SSL and HTTP, but also multiple interoperability protocols, e.g., IIOP and SOAP. The main entities in communication infrastructure include the sender, receiver and transporter. Both senders and receivers are responsible for sending and receiving messages specified by the given interoperability protocols, transforming the messages specific to interoperability protocols into the messages specific to underlying transport protocols (e.g., TCP and HTTP), and managing connections. The sender can build connections, send requests and receive responses at the client side, while the receiver can listen at a given network ad-dress, receive requests and send responses at the server side. The transporter is responsible for sending and receiving messages through the underlying transport protocol. The messages are understandable to senders and receivers but opaque to transporters so that one transport protocol can be employed by multiple interoperability protocols.

Proxy generator can generate stubs and skeletons from the interface definition of the component automatically. Both stubs and skeletons transform the method invocation specific to the client-side programming languages into the message specific to interoperability proto-cols (e.g., GIOP and SOAP), and vice versa. The transformation contains not only the signature but also non-functional constraints of the given interface, i.e., the interface name, operation name, return value, parameters, exceptions and contexts, e.g., security and transaction. The stubs and skeletons are specific to interfaces, i.e., different stub and skeleton for different interfaces. For a given interface, the number of stubs and skeletons is equal to the number of interoperability protocols supported by the implementation of the given interface. For example, if an interface wants to be accessed by IIOP and SOAP, there would be one stub and skeleton for IIOP and another stub and skeleton for SOAP.

Naming service supports components to publish and retrieve different interoperability ad-dresses with the integration of the naming services specific to the interoperability protocols. Because the naming servers of other component frameworks cannot be modified at all, PKUAS integrates them via proxies. At the initialization of PKUAS, the proxies traverse the associated naming servers to retrieve all of the bindings and then register every available name bound with themselves' references, instead of the interoperability addresses of the tar-get components, into PKUAS naming server. In the lookup of naming bindings, the proxies will retrieve the real interoperability address of the target component from the corresponding naming server. On the other hand, an EJB may be accessed by other components through multiple interoperability protocols or transport protocols. PKUAS has to construct and publish multiple addresses into the specific naming servers with the help of the proxies. Then other components can retrieve the valid addresses from their own naming servers and invoke EJBs through their preferred protocols.

4.5. Meta Objects for RSA

To maintain causal connections between RSA and reflective mechanisms discussed previously, a set of meta objects are required, as shown in Figure 7. Briefly, most of MBeans discussed in Figure 4 are responsible for

maintaining platform RSA. The application specific meta objects are derived from common elements in architectural description languages. They organize meta objects for platform RSA to represent application RSA.

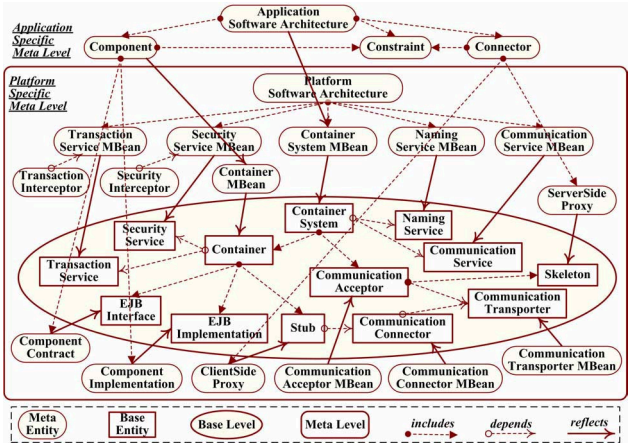

Figure 7. Meta objects and base objects

The operations provided by these meta objects can be divided into five categories: 1) Lifecycle management: the common services, containers and container systems can be started, stopped, suspended, resumed and re-started through manipulating the corresponding elements in platform architecture. 2) Add/remove/replacement: all common services except communication, containers, container systems and application components can be added, replaced and removed at runtime. 3) Statistics: some basic entities may provide some statistics of their internal states and behaviors. Typically, the statistics includes the number of threads in use, the size of buffer or memory footprint and the number of connections for a given application, the number of instances of a given component, the number of invocations for a given method or component, the maximum, minimal and average response time of a given method, and so on. 4) Business invocation: some methods exposed by basic entities can be directly invoked through reflective API. For example, the administrator can explicitly delete some naming bindings from naming service through invoking the 'unbind' method. 5) Basic reflection: all other states and behaviors of the runtime system represented by the elements in platform and application architectures can be observed up-to-date and some of them may be modifiable.

4.6. Programming Model for Reflection

PKUAS provides a reflective programming model for accessing and manipulating its platform RSA and application RSA. As shown in Figure 8, the reflective API is encapsulated in a stateless EJB, called MEJB [19], which brings three advantages. Firstly, reflective mechanisms become secure because EJB is protected by access control mechanism or secure transportation mechanism in EJB container, such as JAAS (Java Authentication and Authorization Service) and IIOP-SSL. Secondly, the users can master the API quickly because it has the same programming model as J2EE. Thirdly, the API can be accessed through multiple interoperability protocols because of PKUAS interoperability framework.

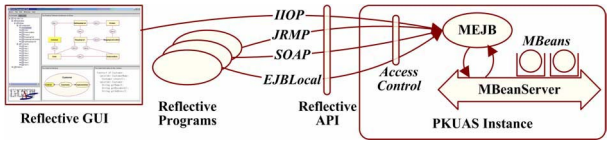

Figure 8. Accessing Reflective Framework

4.7. Performance Evaluation

A test is performed on a PC with PIII 800MHz, 256M SDRAM and Windows 2000 Server with Service Pack 4. It consists of a standalone Java client sending a string in desired bytes and a stateless EJB receiving the string, printing it in screen and returning it back to the client.

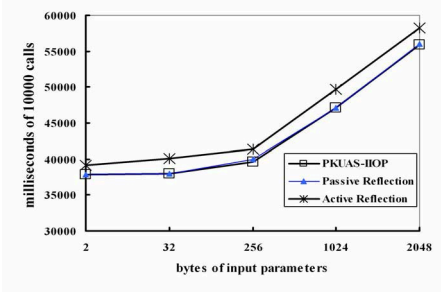

Figure 9. Test result of performance impact

As shown in Figure 9, after RSA is built at runtime, it may perform computation in every invocation (called active) or not (called passive). When RSA is passive, the invocation latency is similar with that of PKUAS without RSA. If RSA is active, the invocation latency increases according to the computation per-formed by RSA. In this test, RSA performs time statistics to expose the minimal, maximum and average response time of the invocations to the specified operation of the EJB. When an invocation comes in, a meta-object increments the invocation counter, records two time-stamps when the invocation comes in and its response goes out, calculates their margin as the response time, determines whether it is minimal or maximum and add it into the total time of all invocations. In this case, the latency increases 3%~5%. Note that, the test EJB is so simple that its computation consumes very little time. In practice, an EJB will be much more complex and take more time to deal with invocations. Consequently,

the percent of time cost in time statistics will decrease. Furthermore, the time statistics can be started and stopped whenever the maintainers need. Then, the performance impact of RSA is reasonable and acceptable in practice.

5. Conclusion and Future Work

Current researches and practices on reflective middleware focus on how to implement reflective mechanisms but pay little attention on the usability and systematic adaptation of reflective middleware. This paper argues that these problems can be well addressed by introducing software architecture into the construction and usage of reflective middleware. To approve this idea, PKUAS, an architecture-based reflective J2EE application server, is presented. PKUAS constructs itself with a set of components based on a micro kernel. Based on such componentized structure, PKUAS implements reflective mechanisms, builds up meta objects to maintain causal connections, and defines a reflective programming model compliant with J2EE. Finally, the result of performance test proves the feasibility and applicability of the architecture-based approach to reflective middleware.

Reflective middleware just provides basic mechanisms to adapt runtime systems. The future work will focus on architecture-based maintenance and evolution that provides the methodology of systematic adaptation and autonomic middleware that can determine why, what and when to adapt without human intervention.

Acknowledgement

This effort is sponsored by the National Key Basic Research and Development Program (973) under Grant No. 2002CB31200003; the National Natural Science Foundation of China under Grant No. 60233010, 60125206; the National High-Tech Research and Development Plan of China under Grant No. 2001AA113060; the Major Project of Science and Technology Research of Ministry Of Education P.R.C. under Grant No. 0214.

References

[1] Agha, G., "Adaptive Middleware: Introduction", *Communications of the ACM*, 2002, 45(6): 30-32.

[2] Blair, G.S., G. Coulson, A. Andersen and etc., "The Design and Implementation of Open ORB 2", *IEEE Distributed Systems Online*, 2001, 2(6).

[3] Cazzola, W., *Communication-Oriented Reflection: a Way to Open Up the RMI Mechanism*, PhD thesis, Milano, Italy, 2001.

[4] Costa, F.M., *Combining Meta-Information Management and Reflection in an Architecture for Configurable and Reconfigurable Middleware*, PhD Thesis, Lancaster University, 2001.

[5] Garlan, D., "Software Architecture: A Roadmap", *The Future of Software Engineering 2000, Proceedings of 22nd International Conference on Software Engineering*, ACM Press, 2000, pp. 91-101.

[6] Gosling, J., B. Joy, G. Steele and G. Bracha, *The Java Language Specification, Second Edition*, 2000.

[7] Hayton, R. and ANSA Team, *FlexiNet Architecture*, Technical Report, 1999.

[8] Huang, G., H. Mei, Q.X. Wang and F.Q. Yang, "A Systematic Approach to Composing Heterogeneous Components", *Chinese Journal of Electronics*, Vol. 12, No. 4, 2003, pp 499-505.

[9] Huang, G., M. Hong and F.Q. Yang, "Runtime Software Architecture based on Reflective Middleware", *Science in China (Series F),* accepted.

[10] Kon, F., M. Roman, P. Liu, J. Mao, T. Yamane, L.C. Magalhaes, and R.H. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB", *In Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing, volume 30, LNCS 1795*, Springer-Verlag, 2000, pp.121-143.

[11] Ledoux, T., "OpenCorba: A Reflective Open Broker", *In Proceedings of the 2nd International Conference on Reflection'99, LNCS 1616*, Springer-Verlag, 1999, pp. 197–214.

[12] Maes, P., "Concepts and Experiments in Computational Reflection", *In Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA' 87)*, Orlando, FL USA, October 1987, pp.147-155.

[13] Mei, H., J.C. Chang and F.Q. Yang, "Software component composition based on ADL and middleware", *Science in China (Series F)*, Vol.44, No.2, 2001, pp. 136-151.

[14] O' Ryan, C., F. Kuhns, D. C. Schmidt, etc., "The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware", *IFIP/ACM Middleware 2000 Conference*, New York, 2000, pp. 154-163.

[15] Shaw, M. and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[16] Smith, B.C., *Procedural Reflection in Programming Languages*, Ph.D Thesis, MIT, 1982.

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

[18] SUN Microsystems, *Java 2 Platform Enterprise Edition Specification, Version 1.3*, 2001.

[19] Sun Microsystems, *Java™ 2 Platform, Enterprise Edition Management Specification*, 2002.

[20] Sun Microsystems, *Java™ 2 Platform, Enterprise Edition Management Specification*，2002.