The Future-Oriented Middleware Technology

Qilin Li
Power System Department, Sichuan Electric Power Test and Research Institute, Chengdu, P.R.China
Email: li_qi_lin@163.com

Mingtian Zhou
School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, P.R.China
Email: mtzhou@uestc.edu.cn

***Abstract*—over the last decade, middleware has emerged as an important architectural component in supporting the construction of distributed applications. However, the current generation of mainstream middleware is always based on the assumptions that distributed applications will run in a static setting. As a result, they fail to provide the appropriate support for the today's networking environment. Major system requirements imposed by today's networking environment are relevant to openness, mobility, and context-awareness. This gives a strong incentive to middleware researchers to investigate the new generation middleware with support for openness, mobility, and context-awareness. In this paper, we first identify the major challenges in open and mobile environments and look deeper into the technical requirements for the future middleware. Subsequently, we introduce reflective computing and reflective middleware to resolve some typical problem, which are related to the future middleware system. Finally, we conclude this paper and point out the future directions of research.**

***Index Terms*—openness, mobility, context-aware, dynamics, uncertainty, adaptability, reconfigurablibility, open coordination, light-weight, heavy-weight, pervasive interoperability, reflective computing, reflective middleware**

I. INTRODUCTION

Over the last decade, middleware has emerged as an important architectural component in supporting the construction of distributed applications. The main purpose of middleware is to shield the heterogeneity of hardware and operating systems and form a new software layer that homogenizes the distributed infrastructure's diversities by means of a well-defined and structured distributed programming model. The role of middleware has proven central to address the ever increasing complexity of distributed systems in a reusable way. Moreover, middleware renders building blocks to be exploited by applications for enforcing non-functional properties, such as scalability, heterogeneity, dependability, performance, resource sharing, and the like. These attractive features of middleware have made it a powerful tool in the software system development practice [1].

Although middleware has well been established and employed, the rapidly evolving computing and networking environments–including new devices and networks, new application domains, and complex associations among them–raise new, challenging requirements such as openness, context-awareness, and increasingly mobility for middleware. Such challenges require a new approach to the design of the next generation middleware [2].

Unfortunately, support for configurability and open engineering is not available in the current generation of mainstream middleware. The main reason is that traditional middleware systems have been built adhering to the metaphor of the black box. Application designers do not have to deal explicitly with problems related to distribution, such as heterogeneity, scalability, resource sharing, and the like. Middleware developed upon network operating systems provides application designers with a higher level of abstraction, hiding the complexity introduced by distribution. Distribution is hidden from both users and application designers, so that the distributed system appears to application developers as a single integrated computing facility. In other words, distribution becomes transparent [3].

Although having proved successful in supporting the construction of traditional distributed systems, transparency philosophy cannot be used as the guiding principle to develop the new abstractions and mechanisms needed by open and configurable middleware to foster the development of the next generation of distributed applications and is suffering from severe limitations when applied to the open and mobile computing settings. Such new challenges require an open implementation approach to the engineering of middleware platforms in terms of being able to configure the underlying support offered by the middleware platform. Besides, it is also important to adopt new approaches allowing inspection and adaptation of underlying components at runtime [2].

In this paper, we attempt to present a view on the future middleware. As part of this view, we identify he major challenges in the today's networking environments and present the technical requirements for the future middleware. In addition, we also introduce reflective computing and reflective middleware to resolve some typical problems.

The remainder of this paper is organized as follows: Section II present the major challenges in open and

doi:10.4304/jcp.5.2.250-257

mobile networking environments. Section Ⅲ identifies a number of requirements for building next generation middleware. In SectionⅣ, we introduces basic concepts related to reflective computing and reflective system. Section Ⅴ defines the concept of reflective middleware and discusses two kinds of reflective methods in the current middleware platforms. Finally, SectionⅥdraws our conclusions and points out the future directions of research.

Ⅱ. MAJOR CHALLEGES IN OPEN AND MOBILE NETWORKING ENVIRONMENT

During the last years, convergence of telecommunication and computer networks, widespread deployment of the Internet, and availability of broadband and wireless networks make network connectivity embedded in most digital devices. With the advent of pervasive networking, resources coordination in distributed systems is not fixed at design time. Future distributed systems will be deployed in an open and mobile networking environment where users can have access to their personal information and public resources anytime and anywhere [1] [5].

In reality, openness is twofold in today's networking environments. On the one hand, Internet connectivity allows networking with resources across network boundaries. On the other hand, wireless networking allows mobile devices held by users to dynamically join and leave networks according to the user's mobility patterns [1]. Such physical mobility can greatly influence network connection, which accordingly has to adapt to user mobility by reconnecting the user with respect to a new location. Mobile devices may interact with different types of networks, services, and security policies as they move from one area from another. This requires applications to behave differently to deal with dynamic changes of the environment parameters [4] [6]. Obviously, such a new environment introduces new technical challenges to middleware designers.

There exist three typical factors that affect the design of the middleware platform required for open and mobile computing environment: mobile device, network connection, and execution context. In a fixed networking environment, devices are fixed, while they are mobile in an open and mobile networking environment. Fixed devices vary from IBM mainframes, to Unix workstations, to home PCs. Mobile devices vary from smartcards, to mobile phones, to digital assistants. While the former are generally powerful machines, with very fast CPU speed and large amount of memories, the latter have limited capabilities, like slow processors speed, little memory, low battery power, small screen size, and so on. It is either impossible or too expensive to augment the resource availability. As a result, middleware should be designed to achieve optimal resource utilization [4] [6].

As for network connection, fixed hosts in a fixed networking environment are usually permanently connected to the network through continuous high-bandwidth links. Disconnections are either explicitly performed for administrative reasons or are caused by unpredictable failures. These failures are considered sporadic and therefore treated as exceptions to the normal behavior of the system. However, such assumptions do not hold for an open and mobile networking environment. In an open and mobile setting, it is often the case that wireless links are relatively unreliable and low-bandwidth. The network connection is characterized by limited bandwidth, high error rate, higher costs, and frequent disconnection owing to power limitations, available spectrum, and mobility. As such, the network connection of open and mobile settings is typically intermittent [4] [5].

In an open and mobile networking environment, execution context means everything that can affect the behavior of an application [5]. It entails resources internal to the device, like amount of memory, screen size, etc.., and external resources, like bandwidth, physical location, quality of the network connection, neighboring services, watches and the like. In a fixed networking environment, context is more or less static: bandwidth is high-speed, continuous and stable. Physical location almost never changes. Although, hosts can be added, deleted or moved, this happens infrequently as they are big, heavy and expensive. Services may change as well, but the discovery of available services is easily performed. In an open and mobile setting, in contrast, context is extremely dynamic and uncertain. Mobile devices may come and leave rapidly, the services may not available when they disconnect from the network and then reconnect. Further, the discovery of service is much more complicated in the mobile settings. In addition, physical location is no longer fixed. Depending on where mobile devices are and if they are moving, bandwidth and quality of network connection may vary greatly as well [2] [3]. In a word, dynamics and uncertainty are intrinsic in the open and mobile networking environments.

Owing to these limitations, traditional middleware technologies designed for a fixed networking environment can not render appropriate support for the open and mobile distributed applications. They generally target a static execution setting where the host location is fixed, the network bandwidth does not fluctuate and services are well defined [4] [5]. As a result, it is necessary to identify a number of important technical requirements that must be provided by the next generation middleware.

Ⅲ. THE TECHNICAL REQUIREMENTS FOR FUTURE MIDDLEWARE

A. Context-awareness

With the network connectivity being pervasive and computing resources being embedded in most objects of our surroundings, context-awareness is becoming a key characteristic of distributed systems. As a result, sensing the physical environment and further adapting the behavior of applications according to both the users' profile and execution context will form a primary concern of the development of distributed systems. Obviously, context-awareness is central to the future middleware. It promotes the usability by a large diversity of users and

enables system deployment and access in open and mobile environments [1].

In general terms, context is the set of environmental states and settings that either determines an application's behavior or in which an application event occurs and is interesting to the user [7]. Specifically, Context information that may influence the behavior of systems can be categorized into three major context domains: environmental context, system context and user context. Environmental context tackles the description of location and of conditions of the physical environment. The system context describes digital, software and hardware resources available to the execution of the system. User context provides information about the users via profiling [1]. In an open and mobile environment, there are two main kinds of context that need to be considered, namely, device context and environmental context. Devices context includes all factors relevant to physical devices in which applications are running, like the mount of memory, power management, screen size, input device, CPU speed, and so on. Environmental context entails any factor except physical devices, including network interface, network bandwidth, communication protocols, current location, etc...

Context-awareness is an important requirement to build an efficient adaptive middleware system. Context-awareness systems have the ability to seamlessly adapt their behavior according to the context within which the systems executes. In an open and mobile setting, the context of a mobile device is usually determined by its current location which, in turn, defines the environment where the computation relevant to the device is performed. The context may include device characteristics, user's activities, services as well as other resources of system. Through context-awareness, the system performance can be improved when execution context is disclosed to the upper layer that assists middleware to tune its own behavior, execute more efficiently, and make the right decisions [4].

B. Adaptability

As stated above, the distributed computing environment now involves a number of technologies with heterogeneous means and requirements. This raises a new important requirement to the future middleware. The future middleware should be context-adaptive and should have the ability to adapt to the current context in order to provide a possibly stable level of service despite changes in the execution conditions [1]

Adaptability allows applications to run efficiently and predictably under a broader range of conditions. By means of adaptability, a middleware system can tune its own behavior instead of providing a uniform interface that caters for the common case. In order to support adaptability, middleware needs to monitor resources, compute adaptation decisions, and notify applications about context changes [4].

Context-based adaptation allows systems to adapt their behaviors according to context changes. The dynamic and uncertain nature of the open and mobile networking environments makes adaptation a necessary technique for context-aware systems. Generally, the range for adaptation strategies is delimited by two extremes. At one extreme, adaptation is entirely the responsibility of individual applications. While this approach avoids the need for system support, it makes context-ware applications more difficult to write. At the other extreme, application-transparent strategy places the whole responsibility for adaptation on the system. This approach does not require changing existing applications. However, the intrinsic limitation of this approach is that there may be situations in which the adaptation performed by the system is adequate or even counterproductive [6]. It could sacrifice performance and functionality. Between these two extremes are referred to as application-aware adaptation. This approach supports collaborative adaptation between the applications and the system. That is, the applications can decide how to best adapt to the dynamically changing context while the system monitors context changes, notifies application of relevant changes and tunes their behaviors.

From system implementation perspective, application-transparent adaptation strategy places the whole responsibility on the middleware layer. In order to provide transparency, middleware must take decisions on behalf of the application; this is inevitably done using built-in mechanisms and policies that cater for the common case rather than the high levels of dynamicity and heterogeneity intrinsic in open and mobile environments. Applications, instead, may have valuable information that could enable the middleware to execute more efficiently, in different contexts [8]. As such, context-aware middleware systems should adopt application-aware adaptation strategy. The middleware has to interact with the application, making the application aware of execution context changes and dynamically tuning its own behavior using information the application passes down in return.

C. Dynamic Reconfigurability and Light Weight System

In an open and mobile environment, application behavior may need to be altered due to the dynamic changes in infrastructure facilities. As a result, dynamic reconfigurability is required in the future middleware and can be achieved by adding a new behavior or changing an existing one at system runtime [4].

Dynamic changes in system behavior and execution context at runtime can trigger reevaluation and reallocation of resources. Middleware capable of supporting dynamic reconfigurability needs to detect changes in available resources and reallocate resources, or notify the application to adapt to the changes [4].

Open and mobile applications often run on resource-scarce devices with slow CPU speed, low amount of memory, limited battery power, and so on. It is not feasible to run heavy-weight middleware systems on these devices due to resource limitations. As a result, it is necessary to choose the right trade-off between computational load and non-functional properties achieved by middleware. Light-weight middleware should be accordingly considered when constructing the future middleware [4] [5].

Traditional middleware platforms like CORBA are too heavy to run on devices with limited resources. They contain a wide range of optional features and all possible functionalities that any application may ever need in order to deal with any kind of problems. This inevitably leads to a computationally heavy middleware system, characterized by large amounts of code and data it use. However, in the most cases, applications use only a small subset of this functionality. Hence, it is necessary to build a lightweight middleware with only a minimal set of functionalities for the future distributed applications [4] [5].

D. Open Coordination Model

The scale of the networking environment within which distributed systems operate, is drastically increasing and further allows coordinating with nodes that are a priori unknown in a possibly very short time period[1]. Such concern has already posed requirements for open and mobile networking environments. Consequently, adequate coordination abstractions needed to be provided by the future middleware.

Classic client-server computing model always assumes that the location of client and server hosts can not change and the connection among them also does not change. Accordingly, the functionality between client and server is statically divided. In an open and mobile environment, however, applications and systems could face wide variations and rapid changes in network conditions and local resources availability. As a result, traditional client-server coordination model is no longer adequate for the open and mobile networking environments.

Open coordination model copes with the problem of high latency and disconnection operations that can arise in other interaction models [4]. Open coordination model allows the interacting parties to be easily added or removed. This type of interaction style thus reduces the network bandwidth consumption, achieves the decoupling of interacting nodes, and improves system salability [4].

E. Pervasive Interoperability

Currently middleware has been introduced to overcome heterogeneity in the underlying hardware and software of distributed systems. By a referencing communication protocol, including message format and distributed interaction model, middleware enables compliant software systems to interoperate [1].

However, interoperability is achieved up to comply with the specific middleware. Further, the emergence of different middleware, to address requirements of specific application domains or networking infrastructures, leads to a heterogeneity issue among communication protocols. It is impossible to predict the communication protocol to be used for accessing networked resources at a given time or at a specific place. As a result, the diversity of communication protocols is a key concern for open and mobile networked environments. The future middleware needs to support the high levels of heterogeneity and interoperability between different middleware platforms.

Ⅳ. REFLECTIVE COMPUTING

Reflection offers significant advantages for building the next generation middleware platforms. Reflection is a principled technique supporting both inspection and adaptation. In traditional middleware, the complexity introduced through distribution is handled by means of abstraction. Implementation details are shielded from both users and application designers and encapsulated inside the middleware itself. However, hiding implementation details means that all the complexity is managed internally by the middleware layer. Middleware is in charge of make decisions on behalf of the application. This may lead to a computational heavy-weight middleware system, which can not run efficiently in a mobile device. Furthermore, in an open and mobile environment, it is neither always possible, nor desirable, to hide all implementation details from applications. Instead, applications may have some valuable information that could lead to more efficient or suitable decisions. Both these limitations can be overcome by reflection. Besides, the possibility opened by reflection is remarkable. Light-weight middleware can be built that support context awareness. Through reflection mechanism, applications can acquire information about their execution context and adapt the middleware behavior accordingly [3].

A reflective system can bring modifications to itself by means of inspection and adaptation. On the one hand, through inspection, the internal behavior of a system is exposed, so that it becomes straightforward to insert additional behavior to monitor the middleware implementation. On the other hand, through adaptation, the internal behavior of a system can be dynamically tuned, by modifications of existing features or by adding new ones [3].As a result, the technique supports more open and configurable middleware.

A. Reflection

The concept of reflection was first introduced by Smith in 1982. In this work, he defined the reflection hypothesis which states [2] [9]:

“In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures”.

The importance of this statement is that a program can access, reason about and alter its own interpretation. Initially, the use of reflection was restricted to the field of programming language design. More recently, Reflection is also increasingly being applied to a variety of other areas including operating system design, concurrent languages and distributed systems [10].

Abstractly, reflection refers to the capability of a system to reason about and act upon itself. More specifically, a reflective system is one that provides a representation of its own behavior, which is amenable to

inspection and adaptation, and is causally connected to the underlying behavior it describes. "Causally-connected" means that changes made to the self-representation are immediately mirrored in the underlying system's actual state and behavior, and vice-versa. It can therefore be said that a reflective system is one that supports an associated causally connected self-representation (CCSR) [11] [12].

B. Reflective Computing

Each reflective computation can be divided into two logical aspects: computational flow context switching and meta-behavior. A computation starts with the computational flow in the base level; when the base entity begins an action, such action is trapped by the meta-entity and the computational flow raises at meta-level (shift-up action). Then the meta-entity completes its meta computation, and when it allows the base-entity to perform the action, the computational flow goes back to the base level (shift-down action) [10] [12].

C. Architectural reflection

In general terms, the architecture of a software system is defined as the system's overall structure as an organized collection of interacting components [13]. It is described by the components that make up the system and how these components are interconnected. Architectural reflection is referred to as computation performed by a system about its own architecture [13]. In an architecturally reflective system, the system architecture is usually reified as a data structure that is causally connected to the actual architecture of the system [14]. This can be used to dynamically examine the architecture of a system at run-time in a structurally reflective manner, but it can also be used to dynamically adapt the architecture of the system as behavioral reflection [15].

D. Base-object, Meta-object and MOP

Just as objects in conventional object-oriented programming language are representations of real world entities, a meta-object is an object that stores information about the implementation and interpretation of its referent (the object it represents) [16]. The set of meta-objects that represent a particular object is that object's meta-level [16]. The set of meta-objects that represent all of the base-objects in an application make up that application's meta-level [15].

More specifically, an object-oriented reflective system is logically structured in two levels. The first level is base-level, which deals with application concerns and describes the computations that the system is supposed to do. The second one is meta-level, which deals with reflective computing and describes how to perform the previous computations. The entities working in the base-level are called base-objects, while the entities working in the meta-level are called meta-objects [15] [17].

Open implementation [18] built around the principle of reflection renders a view of meta interface to the system. Meta interface is separated from tradition interface (base-level) and permits the system to tune its own implementation. The interactions between the base level and meta-level takes place through a set of well-defined interfaces. These interfaces together are known as meta-object protocol (MOP) [17]. The user may use MOP to incrementally modify the implementation and behavior of a system [19]. In an object-oriented system, a MOP can be seen as an extension to the object model, as it specifies which parts of the object model may be reified and possibly changed[15].

E. Metatype

In an object-oriented programming language, an object's type describes the functional behaviors that are directly relevant to the part of the business logic modeled by that object. Generally, metatype is referred to as a characterization of an object's own object model, and as such its non-functional behavior and structure [20]. Examples of metatypes include, but are not limited to, scalability, openness, persistence, heterogeneity, fault tolerance, optimization or resource-sharing [15].

An object's metatype may be orthogonal to its type since the behaviors described in metatypes are not those inherent behaviors of the entity modeled by the object. A metatype can be implemented using meta-objects to implement a non-functional behavior. In reality, objects of a single type may have multiple metatypes associated with them. Similarly several objects of different types may have the same metatype associated with them. Ideally, this association of metatypes should appear transparently to the objects, so that the objects can be written in a manner completely unaware of any metatype that may be applied to it, without changes to the object or its code, and without interrupting any current operation of the object [15].

F. Reification and Absorption

.In all reflective systems, two essential concepts are that of reification and absorption. Reification makes some aspects of the internal representation of the system (the meta-level) explicit and hence accessible from the application (the base-level). Applications are then allowed to dynamically inspect system behavior (inspection), and also to dynamically change it (adaptation), by means of a meta-interface that enables run-time modification of the internal representation previously made explicit. The opposite process where some aspects of the system (the meta-level objects) are altered or overridden is called absorption [8]. Reification and absorption realize the causal connection link of a reflective system.

Ⅴ. REFLECTIVE MIDDLEWARE

The role of reflection in middleware platforms has to do with the introduction of more openness, flexibility and reconfigurability. In the reflective model, the middleware is implemented as a collection of components that can be configured. Like traditional middleware, reflective middleware also render some general services and infrastructures, such as message-passing, remote method call, transaction and component, and the like. In addition, system and application code may also use meta interfaces

to inspect the internal configuration of the middleware and, if needed, reconfigure it to adapt to changes in the environment. As a result, it is possible to select networking protocols, security policies, encoding algorithms, and various other mechanisms to optimize system performance for different contexts and situations [21].

Generally, reflective middleware refers to the use of a causally connected self-representation to support the inspection and adaptation of the middleware system. Thus, the same reflection techniques used in other areas can also apply to middleware. By self-representation, an explicit representation of the internal structure of the middleware implementation that is maintained can be manipulated by it. The self-representation is causally connected if changes in the representation lead to changes in the middleware implementation itself and, conversely, changes in the middleware implementation lead to changes in the representation [21].

In traditional middleware, the complexity incurred by distribution is handled by means of abstraction. Implementations details are shielded from both users and application designers and encapsulated inside the middleware itself. Unlike traditional middleware that is constructed as a monolithic black box, reflective middleware is structured as a group of collaborating components. This organization allows the configuration of very small middleware engines that are able to interoperate with traditional middleware [21]. Conventional middleware implementations include all the functionality that any application may ever need in order to transparently deal with any kind of problems and the solution that guarantees the best quality of service to the application. This may lead to computationally heavy middleware systems, characterized by large amounts of code and data they use. However, in the most cases, applications use only a small subset of this functionality. Hence, it is necessary to build a lightweight and reconfigurable middleware with only a minimal set of functionalities. The possibilities opened by reflection are remarkable. Through reflective mechanisms, the next generation middleware can acquire information about their execution context and tune its own behaviors accordingly [3].

In reflective middleware platforms, two kinds of reflection have been used, namely structural reflection and behavioral reflection [21]. Structural reflection is the ability of a system to provide a complete reification of the application currently executing, for instance, in terms of its methods and state. This enables application developer to inspect or change the functionality of the application and the way it models the domain. Structural reflection renders structural information about the system by means of a concrete representation of (reifying) structural parts of the base level as meta data, such as data structures in the base-level, data types used, inheritance, interfaces implemented, watches and the like. Behavioral reflection is the ability of a system to provide a complete representation of its own semantics, in terms of internal aspects of its runtime environment. This enables application developers to inspect or change the way the underlying environment processes the application, for example, with regard to non-functional properties and resource management. Changing the structure of the base-level system can also be used to change the behavior of that base-level system. Likewise, changing the computation or behavior of the base-level usually involves changing some part the base-level structure of the system [15] [21]. As a result, it is difficult to make a clear distinction between structural reflection and behavioral reflection.

VI. Conclusion and Future Work

Since its emergence, middleware has been proved successful in assisting distributed application development, making development process faster and easier and significantly enhancing software reuse. Middleware layered between the network operating system and the application provides application developers with powerful abstractions and mechanisms that relieve them from dealing with low-level details. The advent of middleware standards, such as CORBA, DCOM and Java/RMI, further promoted the systematic adoption of the technology for distributed application development [1].

The proliferation and development of wireless communication technologies and mobile devices as well as the widespread deployment of Internet have presented new challenges for middleware systems. The diversity and scale of today's networking environments and application domains has made middleware and its association with applications highly complex. Future distributed applications are expected to operate in environments that are highly dynamic and uncertain with respect to resource availability and network [2]. Major system requirements imposed by today's networking environment are relevant to openness, mobility, and context-awareness [1].

However, the principle of transparency that has driven the design of traditional middleware systems may not be optimal for the next generation of distributed applications. In today's open and mobile settings, implementation details have to be made available to the above running applications to allow dynamic reconfiguration of the underlying system, based on different context conditions and varying application needs [10]. In traditional middleware platforms, the degree of support for openness and dynamic configurability does not cover all aspects of the design and the different phases of a platform's life cycle. This is mostly due to the inherent transparent nature of these technologies, which limits the extent to which elements of the design can be opened and exposed to the application developers [21]. In order to provide the best service to the application, the next generation middleware must be aware of its environmental context and current behavior and then tune its own behavior according to setting changes [10]. This gives a strong incentive to middleware researchers to investigate the new generation middleware with support for context-awareness, openness, and mobility. Further, it leads to

leave much work to be done before next generation middleware technology is fully enabled [1] [4].

Reflection is a principled technique that can accommodate the variety of requirements. It offers a truly generic solution to middleware design that naturally renders itself to openness and configurability. Through reflection, future middleware platforms may allow the manipulation and adaptation of the different aspects of its own behaviors in ways that were not anticipated during its design [19].

In this paper, we attempt to explore some key issues related to the future middleware system. In particular, we focus on the major challenges in open and mobile networking environments and look deeper into the technical requirements for the future middleware. Still, we introduce reflective computing and reflective middleware to resolve some typical problems.

Although many research efforts have coped with one aspect or the other of today's networking environment, the research field is still open to further investigation. A first issue that needs to be addressed is how to trade-off between flexibility and performance. Another direction of research concerns dynamically tuning the scope of changes when reconfiguring the system. Some directions of research [1][4] include the representation of context information, open coordination, pervasive interoperability, system security, management of mobile users and proxies, migration from legacy applications to open and mobile middleware platform, etc…

ACKNOWLEDGMENT

The authors would like to acknowledge the anonymous reviewers. This work has been partially funded by National 11-5th High-Tech Support Program of China (2006BAH02A0407).

REFERENCES

[1] Valerie Issarny, Mauro Caporuscio, Nikolaos Georgantas, "A Perspective on the Future of Middleware-based Software Engineering", Future of Software Engineering 2007, L. Briand and A. Wolf edition, IEEE-CS Press. 2007.

[2] G.S. Blair, G.Coulson, P.Robin, M.Papathomas, "An Architecture for Next Generation Middleware", Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake District, UK, pp. 191-206, 15-18 September 1998.

[3] Licia Capra, Wolfgang Emmerich, Cecilia Mascolo, "Middleware for Mobile Computing", UCL Research Note RN/30/01, Submitted for publication, July 2001.

[4] Abdulbaset Gaddah, Thomas Kunz, "A survey of middleware paradigms for mobile computing", Department of Systems and Computer Engineering Carleton University, Tech Rep:SCE-03-16, 2003.

[5] Licia Capra, Wolfgang Emmerich, Cecilia Mascolo, "Middleware for Mobile Computing", UCL Research Note RN/30/01, Submitted for publication, July 2001.

[6] Qilin Li, Wei Zhen, Minyi Wang, Mingtian Zhou, Jun He, "Researches on key issues of mobile middleware technology", Proceedings of the 2008 International Conference on Embedded Software and Systems Symposia （ICESS2008），Chengdu, China, July 2008, IEEE Computer Society，pp.296-301

[7] Guanling Chen, David Kotz, "A Survey of Context-Aware Mobile Computing Research", Dartmouth Computer Science Technical Report TR2000-381, 2000.

[8] Capra, L. and Emmerich, W. and Mascolo, C. (2003) "CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications", IEEE Transactions on Software Engineering, 29 (10). Pp.929-945.

[9] Smith B., Reflection and Semantics in a Procedural Programming Language. PhD thesis Jan. 1982, MIT Press.

[10] Licia Capra,Gordon S.Blair,Cecilia Mascolo, "Exploiting reflection in mobile computing middleware", ACM SIGMOBILE Mobile Computing and Communications Review,2002,10,6(4):pp.34～44.

[11] P. Maes. Concepts and Experiments in Computational Reflection. In Norman K. Meyrowitz, editor, Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87), volume 22 of Sigplan Notices, pages 147–156, Orlando, Florida, USA, October 1987. ACM.

[12] Yang Sizhong, Liu Jinde, Luo Zhigang., "RECOM: A Reflective Architecture of Middleware", Proceedings of International Conferences on Info.-tech.and Info.-net., Beijing, October, 29, 2001, pp.339-344.

[13] W. Cazzola, et al, "Architectural Reflection: Bridging the Gap Between a Running System and its Architectural Specification", in proceedings of 6th Reengineering Forum (REF'98), Firenze, Italy: IEEE. 1998.

[14] J. Dowling , V. Cahill, "The K-Component Architecture Meta-Model for Self-Adaptive Software", Proceedings of Reflection 2001, LNCS 2192, 2001.

[15] John Keeney and Vinny Cahill, "Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework", Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy 2003), Lake Como, Italy, 2003, pp. 3—14.

[16] P. Maes, "Computational Reflection", PhD, Vrije Universiteit Brussels, 1987.

[17] W.Cazzola, "Evaluation of object-oriented reflective model", In Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98), Brussels, Belgium, Jul.1998.

[18] G. Kiczales, "Beyond the Black Box: Open Implementation", in IEEE Software. p. 8-11. 1996

[19] G. Kiczales, J.d. Rivieres, and D. Bobrow, "The Art of the Metaobject Protocol": MIT Press. 1991.

[20] T. Schäfer, "Supporting Metatypes in a compiled, reflective programming language", PhD thesis, Dept. of Computer Science, Trinity College Dublin, Dublin, 131. 2001.

[21] F. Kon, F. Costa, G. Blair, et al, "The case for reflective middleware", Communications of ACM, 2002, 45(6): pp.33～38.

LI Qilin was born in 1973, Chongqing City, China. He received the PhD degree in computer science from University of Electronic Science and Technology of China (UESTC), in 2006. He is now an engineer and team leader in power system department of Sichuan Electric Power Test and Research Institute. His research interests include Dependable Distributed Middleware Systems, Multi-Agent Cooperation Systems and Electric Power Automation.

ZHOU Mingtian was born in 1939, Guangxi Province, China. He received EEBS degree from Harbin Institute of Technology, Harbin, China, in 1962. He became a faculty with UESTC in 1962. He is now a professor and doctoral supervisor of College of Computer Science and Engineering, UESTC, Senior Member of IEEE, Fellow of CIE, Senior Member of FCC, Member of Editorial Board of <Acta Electronica Sinica> and <Chinese Journal of Electronics> and TC Member of Academic Committee of State Council. He has published 13 books and 260 papers. His research interests include Network Computing, Computer Network, Middleware Technology, and Network and Information System Security.