Research on Dependable Distributed Systems for Smart Grid

Qilin Li
Production and Technology Department, Sichuan Electric Power Science 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*— Within the last few years, smart grid has been one of major trends in the electric power industry and has gained popularity in electric utilities, research institutes and communication companies. As applications for smart grid become more distributed and complex, the probability of faults undoubtedly increases. This fact has motivated to construct dependable distributed systems for smart grid. However, dependable distributed systems are difficult to build. They present challenging problems to system designers. In this paper, we first examine the question of dependability and identify major challenges during the construction of dependable systems. Next, we attempt to present a view on the fault tolerance techniques for dependable distributed systems. As part of this view, we present the distributed tolerance techniques for the construction of dependable distributed applications in smart grid. Subsequently, we propose a systematic solution based on the middleware that supports dependable distributed systems for smart grid and study the combination of reflection and dependable middleware. Finally, we draw our conclusions and points out the future directions of research.**

***Index Terms*—smart grid, dependability, dependable middleware, fault-tolerance, fault, error, failure, error processing, fault treatment, replication, distributed recovery, partitioning, open implementation, reflection, inspection, adaptation**

I. Introduction

Within the last few years, smart grid has been one of major trends in the electric power industry and has gained popularity in electric utilities, research institutes and communication companies. The main purpose of smart grid is to meet the future power demands and to provide higher supply reliability, excellent power quality and satisfactory services. Although smart grid brings great benefits to electric power industry, such a new grid introduces new technical challenges to researchers and engineering practioners.

As applications for smart grid become more distributed and complex, the probability of faults undoubtedly increases. Distributed systems are defined as a set of geographically distributed components that must cooperate correctly to carry out some common work. Each component runs on a computer. The operation of one component generally depends on the operation of other components that run on different computers [1] [2]. Although the reliability of computer hardware has improved during the last few decades, the probability of component failure still exists. Furthermore, as the number of interdependent components in a distributed system increases, the probability that a distributed service can easily be disrupted if any of the components involved should fail also increases [2]. This fact has motivated to construct dependable distributed systems for smart grid.

Fault tolerance is needed in many different dependable distributed applications for smart grid. However, dependable distributed systems are difficult to build. They present challenging problems to system designers. System designers must face the daunting requirement of having to provide dependability at the application level, as well as to deal with the complexities of the distributed application itself, such as heterogeneity, scalability, performance, resource sharing, and the like. Few system designers have these skills. As a result, a systematic approach to achieving the desired dependability for distributed applications in smart grid is needed to simplify the difficult task.

Recently, middleware has emerged as an important architectural component in supporting the construction of dependable distributed systems. Dependable middleware can render building blocks to be exploited by applications for enforcing non-functional properties, such as scalability, heterogeneity, fault-tolerance, performance, security, and so on[3]. These attractive features have made middleware a powerful tool in the construction of dependable distributed systems for smart grid [3].

This paper makes three contributions to the construction of dependable distributed systems for smart grid. First of all, we examine the question of dependability and identify major challenges during the construction of dependable systems. Subsequently, we attempt to present a view on the fault tolerance techniques for dependable distributed systems. As part of this view, we present the distributed tolerance techniques for building dependable distributed applications in smart grid. Finally, we propose a systematic solution based on the middleware that supports dependable distributed systems for smart grid and study the combination of reflection and dependable middleware.

The remainder of this paper is organized as follows: Section Ⅱ studies dependability matters for distributed systems in smart grid and identifies the major challenges for the construction of dependable systems. Section Ⅲ introduces basic concepts and key approaches related to fault-tolerance. In SectionⅣ, we discusses distributed fault-tolerant techniques for building dependable systems in smart grid. Section Ⅴ introduces dependable middleware to address the ever increasing complexity of distributed systems for smart grid in a reusable way. Finally, SectionⅥ draws our conclusions and points out the future directions of research.

Ⅱ. DEPENDABLILITY MATTERS

Distributed systems are intended to form the backbone of emerging applications for smart grid, including supervisory control, data acquisition system and distribution management system, and so on. An obvious benefit of distributed systems is that they reflect the global business and social environments in which electric utilities operate. Another benefit is that they can improve the quality of service in terms of scalability, reliability, availability, and performance for complex power systems.

Dependability is an important quality in power distributed applications. In general terms, a system's dependability is defined as the degree to which reliance can justifiably be placed on the service it delivers [4]. The service delivered by a system is its behavior as it is perceived by its user(s); a user is another system (physical, human) which interacts with the former [4]. More specifically, dependability is a global concept that encapsulates the attributes of reliability (continuity of service), availability (readiness for usage), safety (avoidance of catastrophes), and security (prevention of unauthorized handling of information) [2] [4]. In power distributed environments, even small amounts of downtime can annoy customers, hurt sales, or endanger human lives. This fact has made it necessary to build dependable distributed systems for electric utilities.

Fault tolerance is an important aspect of dependability. It is referred to as the ability for a system to provide its specified service in spite of component failure [2] [4]. Fault-tolerant system's behavior is predictable despite of partial failures, asynchrony, and run-time reconfiguration of the system. Moreover, fault-tolerant applications are highly available. The application can provide its essential services despite the failure of computing nodes, software object crash, communication network partition, value fault for applications [5].

However, building dependable distributed systems is complex and challenging. On the one hand, system designers have to deal explicitly with problems related to distribution, such as heterogeneity, scalability, resource sharing, partial failures, latency, concurrency control, and the like. On the other hand, system developers must have a deep knowledge of fault tolerance and must write fault-tolerant application software from scratch [2]. As consequence, they have to face a daunting and error-prone task of providing fault tolerance at the application level [2].

Certain aspects of distributed systems make dependability more difficult to achieve. Distribution presents system developers with a number of inherent problems. For instance, partial failures are an inherent problem in distributed systems. A distributed service can easily be disrupted if any of the nodes involved should fail. As the number of computing nodes and communication links that constitute the system increases, the reliability of components in a distributed system rapidly decreases.

Another inherent problem is concurrency control. System developers must address complex execution states of concurrent programs. Distributed systems consist of a collection of components, distributed over various computers connected via a computer network. These components run in parallel on heterogeneous operating systems and hardware platforms and are therefore prone to race conditions, the failure of communication links, node crashes, and deadlocks. Thus, dependable distributed systems are often more difficult to develop, applications developers must cope explicitly with the complexities introduced by distribution.

In theory, the fault tolerance mechanisms of a dependable distributed system can be achieved with either software or hardware solution. However, the cost of custom hardware solution is prohibitive. In the meantime, software can provide more flexibility than its counterpart[2]. As a result, software is a better choice for implementing the fault tolerance's mechanisms and policies of dependable distributed systems [2]. However, the software solution for the construction of dependable is also difficult. This is particularly true if distributed systems' dependability requirements dynamically change during the execution of an application. Further complicating matters are accidental problems such as the lack of widely reused higher level application frameworks, primitive debugging tools, and non-scalable, unreliable software infrastructures. In that case, fault tolerance can be achieved using middleware [2].

Middleware can be devised to address these problems and to hide heterogeneity and the details of the underlying system software, communication protocols, and hardware. Built-in mechanisms and policies for fault-tolerant can be achieved by middleware and provide solutions to the problem of detecting and reacting to partial failures and to network partitioning. Middleware can render a reusable software layer that supports standard interfaces and protocols to construct a fault-tolerance distributed systems. Dependable middleware shields the underlying distributed environment's complexity by separating applications from explicit protocol handling, disjoint memories, data replication, and facilitates the construction of dependable application [6].

III. Fault Tolerance

A. Failure, Error and Fault

In order to construct a dependable distributed system, it is important to understand the concepts of failure, error, and fault. In a distributed system, a failure occurs when the delivered service of a system or a component deviates from its specification [4]. An error is that part of the system state that is liable to lead to subsequent failure. An error affecting the service is an indication that a failure occurs or has occurred [4]. A fault is the adjudged or hypothesized cause of an error [4].

In general terms, we think that an error is the manifestation of a fault in the distributed system, while a failure is the effect of an error on the service. As a result, faults are potential sources of system failures.

Whether or not an error will actually lead to a failure depends on three major factors. One factor is the system composition, and especially the nature of the existing redundancy [4]. Another factor is the system activity. An error may be overwritten before creating damage [4]. A third factor is the definition of a failure from the user‘s viewpoint. What is a failure for a given user may be a bearable nuisance for another one [4].

Faults and their sources are extremely diversified. They can be categorized according to five main perspectives that are their phenomenological cause, their nature, their phase of creation or of occurrence, their situation with respect to the system boundaries, and their persistence [4].

B. Fault models

When designing a distributed fault-tolerant system, we can not to tolerate all faults. As consequence, we must define what types of faults the system is intended to tolerate. The definition of the types of faults to tolerate is referred to as the fault model, which describes abstractly the possible behaviors of faulty components [2] [4]. A system may not, and generally does not, always fail in the same way. The ways a system can fail are its fault modes. As a result, the fault model is an assumption about how components can fail [2] [4].

In distributed systems, a fault model is characterized by component and communication failures [2] [4]. It is common to acknowledge that communication failures can only result in lost or delayed messages, since checksums can be used to detect and discard garbled messages[2] [4]. However, duplicated or disordered messages are also included in some models [2] [4].

For a component, the most commonly assumed fault models are (in increasing order of generality): stopping failures or crashes, timing fault model, value fault model and arbitrary fault model [2] [4]. Stopping failures or crashes is the simplest and most common assumption about faulty components [2] [4]. This model always assumes that the only way a component can fail is by stopping the delivery of messages and that its internal state is lost [2] [4].

The timing fault model assumes that a component will respond with the correct value, but not within a given time specification [2] [4]. A timing fault model can result in events arriving too soon or too late. A timing fault model includes delay and omission faults [2] [4]. A delay fault occurs when the message has the right content but arrives late [2] [4]. An omission fault occurs when no message is received. Sometimes, delay faults are called performance faults [2] [4]. In the value fault model, the value of delivered service does not comply with the specification [2] [4].

Arbitrary fault model is the most general fault model, in which components can fail in an arbitrary way [2] [4]. As a result, if arbitrary faults are considered, no restrictive assumption will be made [2] [4]. An arbitrarily faulty component might even send contradictory messages to different destinations (a so-called byzantine fault) [2] [4]. This model can include all possible causes of fault, such as messages arriving too early or too late, messages with incorrect values, messages never sent at all, or malicious faults [2] [4].

C. Error Processing and Fault Treatment

Fault tolerance is system’s ability to continue to provide service in spite of faults [2] [4]. It can be achieved by two main forms: error processing and fault treatment [2] [4]. The purpose of error processing is to remove errors from the computational state before a failure occurs, if possible before failure occurrence, whereas the purpose of fault treatment is to prevent faults from being activated again [2] [4].

In error processing, error detection, error diagnosis, and error recovery are commonly used approaches [2] [4]. Error detection and diagnosis is an approach that first identifies an erroneous state in the system, and then assesses the damages caused by the detected error or by errors propagated before detection [2] [4]. After error detection and diagnosis, error recovery substitutes an error-free state for the erroneous state [2] [4].

Error recovery may take on three forms: backward recovery, forward recovery, and compensation [2] [4]. In backward recovery, the erroneous state transformation consists of bringing the system back to a state already occupied prior to error occurrence [2] [4]. This entails the establishment of recovery points, which are points in time during the execution of a process for which the then current state may subsequently need to be restored [2] [4].In forward recovery, the erroneous state transformation consists of finding a new state, from which the system can operate [2] [4]. Error compensation renders enough redundancy so that a system is able to deliver an error-free service from the erroneous state [2] [4].

The goal of fault treatment determines the cause of observed errors and prevents faults from being activated again [2] [4]. The first step in fault treatment is fault diagnosis, which consists of determining the cause(s) of error(s), in terms of both location and nature [2] [4]. Then it takes actions aimed at making it (them) passive [2] [4]. This is achieved by preventing the component(s) identified as being faulty from being invoked in further executions [2] [4]. Fault treatment can be used to reconfigure a system to restore the level of redundancy so that the system is able to tolerate further faults [2] [4].

Ⅳ. Distributed Tolerance Techniques

A. Replication

In order to mask the effects of faults, distributed fault tolerance always requires some form of redundancy. Replication is a classic example of space redundancy. It exploits additional resources beyond what is needed for normal system operation to implement a distributed fault-tolerant service [2] [4]. The metaphor of replication is to manage the group of processes or replicas so as to mask failures of some members of the group [2] [4]. By coordinating a group of components replicated on different computing nodes, distributed systems can provide continuity of service in the presence of failed nodes [2] [4].

There are three well-known replication schemes: active replication, passive replication, and semi-active replication. In active replication scheme, every replica executes the same operations [2] [4]. Input messages are atomically multicasted to all replicas, who all process them and update their internal states. All replicas generate output messages [2] [4].

Passive replication is a technique in which only one of the replicas (the primary) actively executes the operation, updates its internal state and sends output messages [2] [4]. The other replicas (the standby replicas) do not process input messages; however, their internal state must be updated periodically by information sent by the primary [2] [4]. If the primary should fail, one of the standby replicas is elected to take its place [2] [4].

Semi-active replication is a technique which is similar to active replication [2] [4]. In semi-active replication, all replicas will receive and process input messages. However, unlike active replication, the processing of messages is asymmetric in that one replica (the leader) takes responsibility for certain decisions (e.g., concerning message acceptance) [2] [4]. The leader replica can enforce its choice on the other replicas (the followers) without resorting to a consensus protocol [2] [4]. One alternative for semi-active replication is that the leader replica may take sole responsibility for sending output messages [2] [4]. Semi-active replication primarily targeted at crash failures. However, under certain conditions, this strategy can also be extended to deal with arbitrary or byzantine failures [2] [4].

Continuity of service in the presence of failed nodes requires replication of processes or objects on multiple nodes [2] [4]. Replication can provide high-available service for a dependable distributed system. By replicating their constituent objects and distributing their replicas across different computers connected by the network, distributed applications can be made dependable [5]. The major challenge of replication technique is to maintain replica consistency [7] [8] [9]. Replication will fail in its purpose if the replicas are not true copies of each other, both in state and in behavior [5] [10] [11] [12].

B. Distributed Recovery

In a dependable distributed system, some form of recovery is required to minimize the negative impact of a failed process or replica on the availability of a distributed service [4]. In its simplest form, this can be just a local recovery of the failed process or replica. However, distributed recovery will occurs if the recovery of one process or replica requires remote processes or replicas also to undergo recovery [4]. In this case, processes or replica must rollback to a set of checkpoints that together constitute a consistent global state [4].

In order to create checkpoints, there are several major approaches. One way is asynchronous checkpointing [4]. In asynchronous checkpointing, checkpoints are created independently by each process or replica, and then when a failure occurs, a set of checkpoints must be found that represents a consistent global state [4]. This approach aims to minimize timing overheads during normal operation at the expense of a potentially large overhead when a global state is sought dynamically to perform the recovery [4]. The price to be paid for asynchronous checkpointing is domino effect. If no other global consistent state can be found, it might be necessary to roll all processes back to the initial state [4]. As a result, in order to avoid the domino effect, checkpoints can be taken in some coordinated fashion.

Another way is to structure process or replica interactions in conversations [4]. In a conversation, processes or replicas can communicate freely between themselves but not with other processes external to a conversation [4]. If processes or replicas all take a checkpoint when entering or leaving a conversation, recovery of one process or replica will only propagate to other processes or replica in the same conversation [4].

A third alternative is synchronous checkpointing [4] [13]. In this approach, dynamic checkpoint coordination is allowed so that a set of checkpoints can represent global consistent states [4] [13]. As consequence, the domino effect problem can be transparently avoided for the software developers even if the processes or replicas are not deterministic [4]. At each instant, each process or replica possesses one or two checkpoints: a permanent checkpoint (constituting a global consistent state) and another temporary checkpoint [4]. The temporary checkpoints may be undone or transformed into a permanent checkpoint. The creation of temporary checkpoints, and their transformation into permanent ones, is coordinated by a two-phase commit protocol to ensure that all permanent checkpoints effectively constitute a global consistent state [4].

C. Partitioning Tolerance

A distributed system may partition into a finite number of components. The processes or replicas in different components can not communicate each other [11]. Partitioning may occur due to normal operations, such as in mobile computing, or due to failures of processes or inter-process communication. Performance failures due to overload situations can cause ephemeral partitions that are difficult to distinguish from physical partitioning [4].

Partitioning is a very real concern and a common event in wide area networks [4]. If the network partitions, different operations may be performed on the processes or replicas in different components, leading to inconsistencies that must be resolved when communication is re-established and the components remerge [5]. One strategy for achieving this is to allow components of a partition to continue some form of operation until the components can re-merge [4] [11]. Once the components of a partitioned remerge, the processes or replicas in the merged components must communicate their states, perform state transfer and reach a global consistent state [5].

As another example, certain distributed fault-tolerance techniques are aimed at adopting dynamic linear voting protocol to ensure replica consistency in partitioned networks [5]. Voting protocols are based on quorums. In voting protocols, each node is assigned a number of votes. When a network is partitioned or remerged, if a majority of the last installed quorum is connected, a new quorum is established and updates can be performed within this partition [5].

V. DEPENDABLE MIDDLEWARE

In the past decade, middleware has emerged as a major building block in supporting the construction of distributed applications [14]. The development of distributed applications has been greatly enhanced by middleware. Middleware provides application developers with a reusable software layer that relieve them from dealing with frequently encountered problems related to distribution, such as heterogeneity, interoperability, security, scalability, and so on[14][15][16][17]. Implementation details are encapsulated inside the middleware itself and are shielded from both users and application developers', so that the infrastructure's diversities are homogenized by middleware [18] [19] [20] [21]. These attractive features have made middleware an important architectural component in the distributed system development practice. Further, with applications becoming increasingly distributed and complex, middleware appears as a powerful tool for the development of software systems [14].

Recently, a strong incentive has been given to research community to develop middleware to provide fault tolerance to distributed applications [2]. Middleware support for the construction of dependable distributed systems has the potential to relieve application developers from the burden by making development process faster and easier and significantly enhancing software reuse. Hence, such middleware can render building blocks to be exploited by applications for enforcing dependability property [2].

However, building such software infrastructure that achieves dependable goal is not an easy task. Neither the standard nor conventional implementations of middleware directly address complex problems related to dependable computing, such as partial failures, detection of and recovery from faults, network partitioning, real-time quality of service or high-speed performance, group communication, and causal ordering of events[9]. In order to cope with these limitations, many research efforts have been focused on designing new middleware systems capable of supporting the requirements imposed by dependability [5].

A first issue that needs to be addressed by dependable middleware is interoperability [2]. Interoperability allows different software systems to exchange data via a common set of exchange formats, to read and write the same file formats, and to use the same protocols. As a result, in order to be useful, dependable middleware should be interoperable [2]. Through interoperability, dependable middleware can provide a platform-independent way for applications to interact with each other [2]. In other words, two systems running on the different middleware platforms can interoperate with each other even when implemented in different programming languages, operating systems, or hardware facilities [2].

Another important problem concerns transparency. Dependable middleware should provide some form of transparency to applications [2]. It allows dynamically to add to an existing distributed application and to interfere as little as possible with applications at runtime. Therefore, many existing applications can benefit from the dependable middleware [2]. Traditional middleware is built adhering to the metaphor of the black box. Application developers do not have to deal explicitly with problems introduced by distribution. Middleware developed upon network operating systems provides application developers with a higher level of abstraction. The infrastructure's diversities are hidden from both users and application developers, so that the system appears as a single integrated computing facility [16].

Although transparency philosophy has been proved successful in supporting the construction of traditional distributed systems, it cannot be used as the guiding principle to develop the new abstractions and mechanisms needed by dependable middleware to foster the development of dependable distributed systems when applied to the today's computing settings[15][18][19]. As a result, it is important to adopt an open implementation approach to the engineering of dependable middleware platforms in terms of allowing inspection and adaptation of underlying components at runtime[22][23][24][25].

With networks becoming increasingly pervasive, major system requirements posed by today's networking infrastructure relate to openness and context-awareness [14]. This leads to investigate new approaches for middleware with support for dependability and context-aware adaptability. However, in order to provide transparency, traditional middleware must make 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 today's networking environments[16][19]. The application, however, can normally make more efficient and better quality decisions based on application-specific information that could enable the middleware to execute more efficiently, in different contexts[16][19]. As such, it is not appropriate to place the whole responsibility for adaptation on the dependable middleware [16] [19]. Dependable middleware, instead, may interact with the application, making the application aware of execution context changes and dynamically tuning its own behavior using valuable application information [16] [19].

Reflection offers significant advantages for building dependable middleware in the today's computing settings [26] [27] [28] [29] [30]. Reflection is a principled technique supporting both inspection and adaptation [26] [27] [28] [29] [30]. A reflective dependable middleware system can bring modifications to itself by means of inspection and adaptation [26] [27] [28] [29] [30]. On the one hand, through inspection, the internal behavior of dependable middleware 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 dependable middleware can be dynamically tuned, by modifications of existing features or by adding new ones [26] [27] [28] [29] [30]. As consequence, the reflection technique can support more open and configurable dependable middleware. Reflection mechanism can enable dependable middleware systems to inspect or change the way the underlying environment processes the application [31] [32]. Through reflection mechanism, dependable middleware systems can acquire information about their execution context and adapt their behaviors accordingly [31] [32].

In addition to being application-transparent, dependable middleware also needs to provide a simple interface that allows applications to specify desires about the dependability and to provide automatic detection of and recovery from faults [2]. Besides, when the dependability requirements dynamically change at runtime, the dependability mechanisms may change during the execution of an application [2]. Therefore, dynamic reconfigurability is also required in the dependable middleware. Dynamic reconfigurability can be achieved by adding a new behavior or changing an existing one at system runtime. Dependable middleware capable of supporting dynamic reconfigurability needs to detect changes in dependability requirements and the faults that occur in the environment, and reallocate resources, or notify the application to adapt to the changes [2].

Ⅵ. Conclusion and Future Work

Over the few years, increasing attention has been given to smart grid. As a new paradigm in the power grid, smart grid exploits the latest information and communication technologies to accommodate renewable energy generation and to construct smart measurement system, demand-side response, and distribution automation to transmission grid intelligence [33][34][35][36][37][38]. With the advent of smart grid era, electric power systems are confronted with new challenges. The diversity and scale of networking environments and application domains has made smart grid and its association with applications highly distributed and complex. Future smart grid applications are expected to operate in environments that are highly distributed and dynamic with respect to resource availability and network. As consequence, the likelihood of faults undoubtedly increases. This gives a strong incentive to researchers and engineering practioners to investigate the construction of dependable distributed systems. Further, it leads to leave much work to be done before smart grid technology is fully enabled.

In this paper, we attempt to explore some key issues related to building a dependable distributed system for smart grid application. In particular, we focus on the dependability matters, major challenges and distributed tolerance techniques for the construction of dependable systems. Still, we also look deeper into the systematic approach to providing the dependable support for smart grid applications. In addition, we introduce basic concepts and techniques for fault-tolerance. While some of the insights might seem rather intuitive in hindsight, we think that these views are often sadly neglected in the development of dependable distributed applications. It is our sincere hope that dependable middleware implementers and application developers will benefit from our knowledge and contributions and that our insights will help to shape the future of dependable infrastructures for middleware and distributed applications.

Although existing research efforts have addressed some issues related to the construction of the dependable distributed systems for smart grid, many issues require further investigation. Some open issues, such as combining dependability and real-time, combining fault tolerance and security, combining replication and group communication, combining legacy applications and dependable middleware, still remain to be addressed by the developers of dependable distributed systems for smart grid[5][11].

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) and key Technology Support Program of the State Grid of China (WG1-2010-X).

References

[1] [1] 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. 333 - 338

[2] [2] J. Ren, AQuA: “A Framework for Providing Adaptive Fault Tolerance to Distributed Applications”, PhD thesis, University of Illinois at Urbana-Champaign, 2001

[3] V [3] 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

[4] C.Laprie, editor, “Dependability: Basic Concepts and Terminology”, Springer-Verlag, Vienna, 1992

[5] Qilin Li, Wei Zhen, Mingtian Zhou “Middleware for Dependable Computing”, Proceedings of the 2008 International Conference on Embedded Software and Systems Symposia（ICESS2008）, Chengdu, China, July 2008, IEEE Computer Society，pp.296-301

[6] Kurt Geihs, “Middleware challenges ahead”, IEEE Computer, June 2001, 34(6): 24—31

[7] S.Krishnamurthy, “An Adaptive Quality of Service Aware Middleware for Replicated Services”, PhD thesis, University of Illinois at Urbana-Champaign, 2002

[8] P.Narasimhan, “Transparent Fault Tolerance for CORBA”, PhD thesis, University of California at Santa-Barbara, 1999

[9] S.Maffeis, D.C.Schmidt, “Constructing Reliable Distributed Systems with CORBA”, IEEE Communications Magazine, 35(2): pp.56-60, Feb.1997

[10] P.Felber, “The CORBA Object Service: a Service Approach to Object Groups in CORBA”, PhD thesis, Swiss Federal Institute of Technology at Lausanne, Switzerland, 1998

[11] P.Felber, P.Narasimhan, “Experiences, Strategies, and Challenges in building Fault-Tolerant CORBA Systems”, IEEE Transactions on computers, 53(5):pp.497-511, May.2004

[12] B.Natarajan, A.Gokhale, S.Yajnik, D.C.Schmidt, “DOORS: Towards High-performance Fault Tolerance CORBA”, in Proceedings of the 2nd Distributed Applications and Objects(DOA) conference, Antwerp, Belgium, Sep. 21-23, 2000

[13] E.N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems", in ACM Computing Surveys, 34(3):pp.375-408, September 2002

[14] 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

[15] 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

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

[17] 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

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

[19] 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

[20] 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

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

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

[23] 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

[24] 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

[25] 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

[26] P. Maes, “Computational Reflection”, PhD, Vrije Universiteit Brussels, 1987

[27] 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

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

[29] G. Kiczales, J.D. Rivieres, and D. Bobrow, “The Art of the Metaobject Protocol”: MIT Press. 1991

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

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

[32] 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

[33] Zhong Jin，Zheng Rui-min，Yang Wei-hong，Felix Wu, “Construction of Smart Grid at Information Age”, Power System Technology, 2009，33 (13). Pp.12-18(in Chinese)

[34] Research Reports International, “Understanding the smart grid”, RRI00026

[35] The National Energy Technology Laboratory, “Modern grid benefits”, Pitt sburgh , PA , USA : NETL , 2007

[36] The Electricity Advisory Committee, “Smart grid：Enabler of the new energy economy”[EB/OL], 2008-12-01[2009-04-20]

[37] Jing Ping，Guo Jian-bo，Zhao Bo，Zhou Fei，Wang Zhi-bing, “Applications of Power Electronic Technologies in Smart Grid”, Power System Technology, 2009，33 (15). Pp.1-6(in Chinese)

[38] Zhang Wen-liang，Liu Zhuang-zhi，Wang Ming-jun，Yang Xu-sheng, “Research Status and Development

Trend of Smart Grid ” , Power System Technology, 2009，33 (13). Pp.1-11(in Chinese)

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 vice director in production and technology department of Sichuan Electric Power Science and Research Institute. His research interests include Smart Grid, Electric Power Automation Dependable Distributed Middleware Systems, and Multi-Agent Cooperation Systems.

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.