A Brief Survey of Event-based Middleware

Tao Wang, Wenbo Xu
School of Information Technology
Jiangnan University
Wuxi, China，214122
e-mail: wwtt263_cn@sina.com, xuwb_sytu@yahoo.cn

Jianli He，Rong Chen, Weinan Gu
System Software Engineering Centre
Tongji University
Shanghai, China，200092
e-mail: Hejianli74@gmail.com

***Abstract*—The event-based middleware is well adapted to develop the adaptive application for the dynamic large-scale distributed computing environment and is getting more and more popular in both the academy and the industry due to its asynchronous, one-to-many and loosely-coupled push communication properties. This paper combs the concepts and architecture of event-based middleware. We briefly present five influential event-based research results, and explore the research efforts and core contents. Our research results that show the paradigm is also valuable for developing general-purpose applications are also introduced from the perspective of architecture, implementation and application.**

Keywords-event-based middleware; adaptive middleware architecture; GUI system implementaion

I. INTRODUCTION

With the development of communication technologies and computers, all kinds of computers are gradually utilized to the people's living and working space. At the same time, some new computing paradigms are emerging, such as pervasive computing and cloud computing. These computing paradigms have the common natures that involve thousands of entities whose location and behavior may greatly vary throughout the life-time of the system. The middleware sitting between the networking operating system and industry-specific applications is to solve the heterogeneity and distribution problems by offering distributed system services that have standard programming interfaces and protocols [1]. While, many existing middleware do not provide adequate support to meet the highly dynamic distributed computing demands. Tree results about the limitations of many existing middleware are summarized by [2]: many existing distributed systems, such as distributed file systems and remote procedure call, cause great loss in service quality and failure resilience due to its hiding distribution and masking remote resources as local resources; Composing distributed applications through programmatic interfaces leads to a tight coupling between major application components, so that the system's extensibility and scalability is restricted in; distributed object systems that encapsulate both data and functionality within objects go against the sharing, searching, and filtering of data.

The event-based middleware is well adapted to develop the adaptive application for the dynamic large-scale distributed computing environment and is getting more and more popular in both the academy and the industry due to its asynchronous, one-to-many and loosely-coupled push communication properties. The partner in event-based systems includes subscribers and publisher. Subscribers express their interest in an event, or a pattern of events. Messages related to the event generated by a publisher are asynchronously propagated to all subscribers that registered the given event. The attraction of event-based system lies in the full decoupling in time, space, and synchronization between publishers and subscribers [3], space decoupling means that the interacting parties do not need to know each other's identities, time decoupling refers to the interacting parties need not be active at the same time when they need to be participating in the interaction, and synchronization decoupling means that the interacting parties don't block each other's activity in the course of exchanging messages. On the other hand, comparing to asynchronous push, synchronous poll has following limitations [4]: synchronous method invocation is insufficient for reactive environments; Too-frequent polling burdens the system and too-infrequent polling delays the communication.

Firstly, this paper combs the concepts and architecture of event-based middleware. Then, we briefly present five influential event-based research results, and explore the research efforts and core contents. Lastly, our research results are also introduced from the perspective of architecture, implementation and application.

II. ARCHITECTURE AND CONCEPTS

Events are occurrences that are localized in time and space, such as the invocation of a program, the modification of a file, the change of a participant's state, or the sending of message [5, 6]. Messages are information emitted by some partner in order to communicate an event. Fig. 1 describes the abstract architecture of event-based systems. This paradigm usually called "publish/subscribe" according to the names of the two basic operations that initiate the communication. Although many references name the architecture as many-to-many communication model, we call it one-to-many model from the publishers and subscribers perspective where the messages published by one publisher can be received by many subscribers. The publisher and subscriber are autonomous entities that perform application-specific tasks. An event is generated by a publisher and the messages related to the event are emitted by the publisher. Event broker provides storage and management ability for subscriptions and efficient delivery of events. Event broker is a neutral mediator between publishers and subscribers. Such an event broker waits for the occurrence of an event, receives event publisher's messages, and delivers them to those subscribers that have explicitly declared their interest in receiving it. The event broker plays a key role in the loosely-coupled system characteristics.

Since standard interfaces are important to middleware platform, the event-based middleware should at least provide four basic operations. The subscriber who needn't know the event publishers' identities register their interest in events by a subscribe() operation on the event broker. The subscription information remains only stored in the event broker and the event publisher needn't keep those information. The symmetric operation unsubscribe() applies for terminating a subscription. When a event occurs, a publisher calls a publish() operation to convey the messages related to the event to the event broker. Once receives messages from publisher, the event broker call the notify() operation to propagate the event occurrence information to all relevant subscribers.

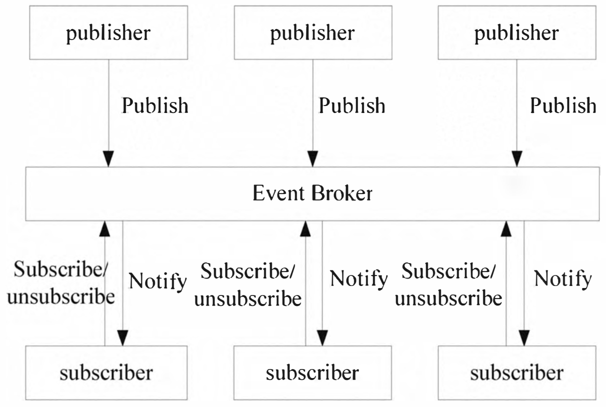

Figure 1. Abstract architecture of event-based system

III. RESEARCH RESULTS

Reference [7] proposes a comprehensive framework for the construction of large-scale, event-based software systems for the Internet, in order to captures event-based systems' design dimensions. The framework comprises seven models: an object model, an event model, a naming model, an observation model, a time model, a notification model and a resource model. This section does not touch in detail every aspects of this framework, but briefly present the main research results. Based on the achievements, the recent research areas in event-based systems include mobility support, dynamic and peer-to-peer systems, formal modeling and security.

CORBA is an object-oriented middleware architecture designed by the OMG, the largest consortium in the computing industry. The CORBA Event Service [8] is centered on the event channel, which maintains a contact between publishers and subscriber in a decoupled fashion. The communication among publishers and subscribers can be both in the pull and in the push model. While, the asynchronous communication has the substantial overhead for event delivery and no event filtering is supported. The CORBA Notification Service[9] try to overcome the shortcomings of the Event Service by providing event filtering, quality of service, and a lightweight form of typed and structured events.

The CEA addresses the emerging need for asynchronous communication in multimedia and sensor-rich application in the early 90s. "The CEA supports asynchronous operation by means of events, event classes, and event occurrences as object instances [10]". The event architecture includes event sources, sinks and event mediators. Event filtering based on parameter templates happens at the event sources to reduce communication overhead. The tight coupling between event sources and sinks is decoupled by event mediators. The CEA provides a language and composition operators for specifying composite events and use stream semantics to model event arrival from the different sources.

JEDI [11] is an object-oriented infrastructure that supports the development and operation of event-based systems. A system consists of active objects and event dispatchers, the former publish or subscribe to events, and the later route events. JEDI provides the centralized and distributed implementations of the event dispatcher. The distributed version organizes event dispatchers in a tree structure. The tree is built dynamically as a core-based tree [12]. The core named a group leader make a global broadcast when it presents. The system also supports mobile computing by offering the moveOut and moveIn operations. JEDI chose to guarantee only a particular form of partial ordering among events as far as preservation of event ordering is concerned.

Siena [13, 14] is one of the first implementations of an Internet-scale event notification service developed at the University of Colorado. Siena focuses on the balances between expressiveness and scalability, and explores content-based routing in a wide-area network. A set of event brokers cooperating with each other form a network-wide event service. Siena supports several network topologies consisting of multiple brokers, including hierarchical, acyclic peer-to-peer, and general peer-to-peer topologies. Event publishers and subscribers are clients that first need to connect to an event broker in the network of brokers. Although Siena is a promising approach for a large-scale middleware, there exist some problems that should be resolved, including [14]: lacks support for type-checking; the topology of the overlay network of event brokers is static; the scalability is limited in some case.

Hermes [15, 16] is a distributed event-based middleware platform. To handle dynamic large-scale environments, Hermes uses peer-to-peer techniques based on an overlay network called Pan that supports a variant of the advertisement semantics. The features of the underlying overlay system for message routing, scalability, and improved fault-tolerance are especially emphasized. The routing algorithms use rendezvous points to reduce routing state. Hermes places particular emphasis on programming language integration through following a type- and attribute-based publish/subscribe model.

In this section, we briefly present the CORBA event service, CEA, JEDI, Siena and Hermes. There are some other influential event-based systems such as Elvin, Rebeca and Gryphon etc. The interested reader refers to [17].

IV. Our Approaches

Armed with the event-based features, we have developed an adaptive middleware platform. Differing from the middleware mentioned above, the platform is general purpose for developing adaptive application.

A. *Architecture*

Fig. 2 illustrates the platform architecture. The architecture [18] consists of reflection meta-level and application level. Meta-level provides the system supports for developing adaptive application, consisting of three independent models, namely the interface model, the assembly model and the perception model. The meta-level interface model encapsulates reusable component in the form of binary code and exposes interfaces of the component to its clients. The interface model separates application from reusable components that make the reusable components dynamically loadable and replaceable, which is very important for developing adaptive application. The assembly meta-model classifies components and separate the identifiable units of system functionality from the user viewpoint from the whole system functionality. These functional units can be encapsulated into feature component class, and a running object can dynamically aggregate feature class object. Furthermore, allowing for component evolution by itself, the assembly model allows component classes extending. Based on event-based technologies, the perception meta-model serves for the data exchange between objects and provides execution environments for running applications.

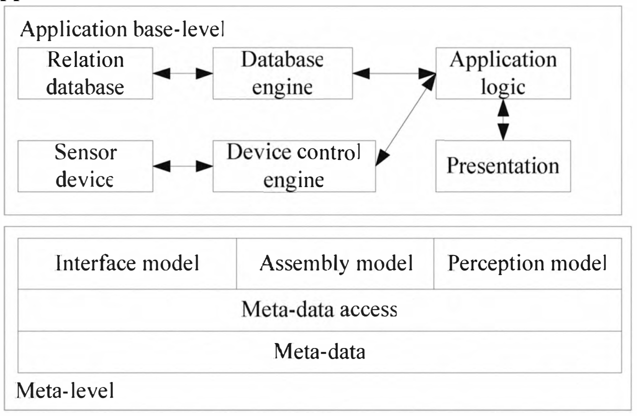

Figure 2. Adaptive middleware architecture

Separation of concern is not particularly novel for software practitioners. The middleware platform especially contributes to the strict separation of concern between application and presentation. The presentation transforms the application's state into a perceivable (i.e., graphical and audible) representation at application base-level. The relation database stores application persistent states.

B. *Implementation of perception model*

The perception model addresses data-sharing between the server component object and his clients. Applications usually include lots of server component object, each owning a service broker. Server object provides his clients with the ability to express their interest in an event, in order to be notified subsequently of inner state change. The broker records all active subscriptions of registered clients. When an event happens, it would be asynchronously propagated by service broker to all subscribers that registered interest in that given event.

In our architecture, events are produced by component objects and consumed by any interested parties. Event and messages type are defined in component description language (CDL), and the data object propagating between partners is self-description type. CDL plays three important roles: (1) defines event type and checks grammars; (2) binds component object with event broker; (3) generators automatically application programming interfaces.

The clients of a component object register their interest in events by calling a SubscribeType() operation on the server object. This subscription information remains stored in the event service broker, until the symmetric operation UnsubscribeType() is invoked. Each subscription has a handler parameter responding to event notification. To generate an event, the server object calls a PublishType() operation. The event broker propagates the event to all relevant subscribers through a NotifyType() operation. Asynchronous notification immediately returns to server object's control flow and doesn't care about the execution of event handler function. Then, synchronous notification blocks the server object before client object' handler has finished. Each event support an information space associated with an event schema. Event filtering reduces the number of events from the server object to his clients by matching events against a template. Filtering can be done at the event source. When an event filter is necessary, a function associated with a template is invoked before the event notification is emitted by service broker.

C. *Application*

To provide an execution environment for running applications, we define a new component type named applet. An applet component object is a stateless event processor, consisting of event queue, event scheduler and a number of application-specific server components objects.

An application corresponding to an applet component, usually including some graphic user interface (GUI) components and application data related to persistent state. Fig. 3 shows the GUI system implementation and the interaction between graphic element and application. Events may be generated by clock or input devices, such as mouse or keyboard. The kernel thread collects the events produced by peripherals and stores these events in the event queue. The graphic element form is parent window of most of the other graphic elements, such as the button and menu. When the form object is instantiated in an application, it generates a thread that shares the event queue and dispatches events. The event scheduler loops continuously, processing events in the application queue that stores different events coming from server objects.

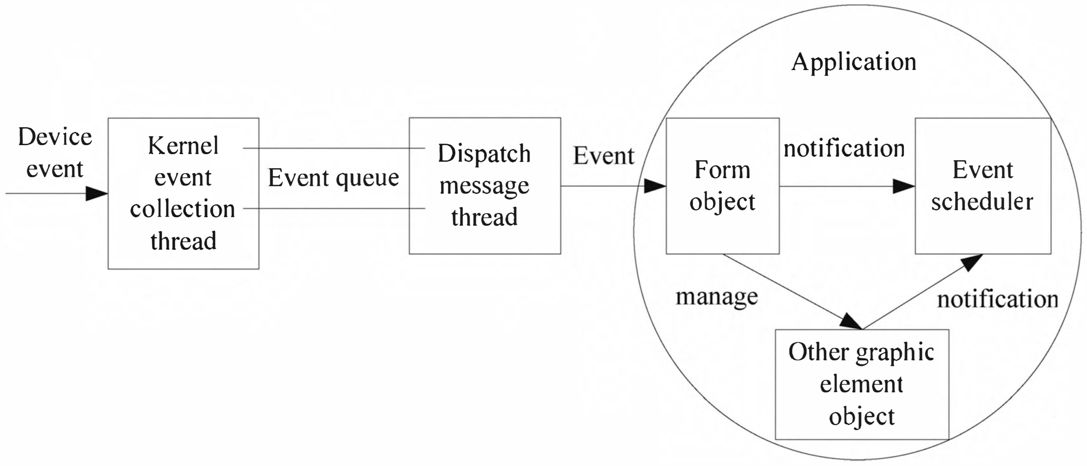

Figure 3. GUI system implementation

V. CONCLUSIONS

This paper presents the event-based middleware that are well adapted to develop the adaptive application for the dynamic large-scale distributed computing environment. Our approaches are introduced. Our platform is general-purpose, and builds on components-based, reflection and event-driven technologies. We have achieved several mobile telephones projects on the platform. Our experiments show that the publish/subscribe paradigm is also well valuable for developing general-purpose applications.

REFERENCES

[1] P. A. Bernstein, "Middleware: A model for distributed system services," Communications of the ACM Vol.39, No.2, 1996, pp. 86–98.

[2] R. Grimm, J. Davis, E. Lemar, et al, "System Support for Pervasive Applications," J. ACM TRANSACTIONS ON COMPUTER SYSTEMS. Vol. 22 (4), 2004, pp. 421–486.

[3] P. T. Eugster, P. A. Felber, R. Guerraoui, and A. M. Kermarrec, "The many faces of publish/subscribe," J. ACM Computing Surveys, Vol.35 (2), 2003, pp. 114–131.

[4] J. Bacon, K. Moody, J.Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, M. Spiteri, "Generic support for distributed applications," IEEE Computer, 33(3), March 2000, pp. 68–76.

[5] D. J. Barrett, L. A. Clarke, P. L. Tarr, and A. E. Wise, "A framework for event-based software integration," ACM Trans. on Software Engineering and Methodology, 5(4) , October 1996, pp. 378–421.

[6] C. Szyperski, "Component Software: Beyond Object-Oriented Programming," Addison-Wesley, 2002.

[7] D. Rosenblum, and A. Wolf, "A Design Framework for Internet-Scale Event Observation and Notification," In Proceedings of the 6th European Software Engineering Conference/ACM SIGSOFT 5th symposium on the foundations of Software Engineering. ACM Press, New York, NY, 1997, pp. 344–360.

[8] OMG. CORBA:Event Service, Version 1.0. Specification, Object management Group(OMG), March 1995.

[9] OMG. CORBA:Notification Service, Version 1.0.1. Specification, Object Management Group(OMG), August 2002.

[10] J. Bacon, K. Moody, J. Bates, R. Hayton, C Ma, A. McNeil, O. Seidel, and M. Spiteri, "Generic support for distributed applications," IEEE Computer, 33(3), 2000, pp. 68–76.

[11] G. Cugola, E. D. Nitto, and A. Fugetta, "The Jedi event-based infrastructure and its application to the development of the opss wfms," IEEE Trans. Softw. Eng. 27, 9 (Sept.), 2001, pp. 827–850.

[12] T. Ballardie, P. Francis, and J. Crowcroft, "Core Based Trees(CBT)," In Procddings of ACM SIGCOMM'93, San Francisco, CA, USA, September 1993.

[13] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and Evaluation of Wide-Area Event Notification Service," ACM Transactions on Computer Systems, 19(3) , August 2001, pp. 332–383.

[14] A. Carzaniga, "Architectures for an Event Notification Service Scalable to Wide-Area Networks," PhD thesis, Politecnico di Milano, Milano, Italy, December 1998.

[15] P. Pietzuch, "Hermes: A scalable event-based middleware," Ph.D. Thesis, Computer Laboratory, Queens'College, University of Cambridge, February 2004.

[16] P. Pietzuch and J. Bacon, "Hermes: A distributed event-based middleware architecture," In Proceedings of the 1st International Workshop on distributed event-based system (DEBS'02), 2002.

[17] S. Tarkoma and K. Raatikainen, "State of the Art Review of Distributed Event Systems," Technical Report C0004, University of Helsinki, 2006.

[18] Jianli He, Rong Chen, and Weinan Gu, "An Adaptive Middleware Platform," Proceeding of the International Conference on Advanced Computer Theory and Engineering, 2009, pp.237–244.