An Approach to Distributed Component-based Real-time Application Software Development

Stephen S. Yau and Bing Xia
Computer Science and Engineering Department
Arizona State University
Tempe, AZ 85287, U.S.A.
{yau, xia}@asu.edu

Abstract

Component-based software development would allow application software be largely constructed, rather than programmed. This approach would dramatically improve the productivity of software development. Although there are many reusable software packages available, the integration of the chosen parts remains to be a very difficult problem because there are many barriers of integration, including programming languages, operating systems, communication mechanism, interface, etc.

In this paper, an approach to developing real-time application software based on a distributed component architecture and cross-platform and cross-language integration of these software components are presented. The Common Object Request Broker Architecture (CORBA) will be used in the implementation. Our distributed components will satisfy easy retrieval and integration over a heterogeneous distributed system environment. A component replication mechanism is used for providing fault-tolerance feature. Using object adapters with real-time request monitor and scheduler that are transparently generated by a distributed component integration tool, real-time and fault-tolerance features can be easily incorporated in the application software.

Keywords: Component-based software development, real-time application software, distributed system, fault-tolerance, CORBA.

1. Introduction

Component-based software is a desirable concept in constructing large-scale applications software [1]. By reusing well-developed software parts to construct application software, the productivity of software development can be dramatically improved. Furthermore, there is a continuous need to upgrade and reintegrate the existing real-time software systems. For component-based software, this can be achieved by replacing some parts of the system with new components with compliant interface and incorporating new technology inside. Comparing this approach to traditional re-design and re-coding methods, the cost for maintenance and upgrade will be reduced dramatically. However, component-based software development only recently appears to be feasible. As the object-oriented software development approach becomes more mature [1][2], it helps the software developers solve some problems in component-based software development such as component decomposition and interface definition. By revising the objects' interfaces and properties in the software components, applications can be largely constructed, rather than programmed directly at the object level.

There are already many reusable software packages available [3]. For example, we can directly select and use a toolbar or a text editor when we develop a GUI software in any visual programming environment without knowing its implementation details [4][5]. Recently, two new component specifications, Sun's JavaBeans [6] and Microsoft's ActiveX Control [4][7][8], have expanded the reuse paradigm to a wider range of software development, such as compound documentation and Internet web services [4]. But, the significant problem of the chosen parts not fitting well together still remains [9] because there are many barriers including programming languages, operating systems, communication mechanisms, interfaces, etc [10]. Furthermore, all these specifications do not address real-time guarantees. To overcome these barriers, we need a common distributed object-oriented environment which offers a consistent view of service access and covers most of the differences from various operating systems, networks, and programming languages. There are two popular

distributed object environments: Microsoft's Distributed Component Object Model (DCOM) [7] and Object Management Group (OMG)'s Common Object Request Broker Architecture (CORBA) [11][12]. We will use the CORBA environment because it is more widely available and allows the integration of a variety of object systems through its standard Interface Definition Language (IDL) [11][13]. By separating object interface from its implementation, it facilitates independently developed components with compatible IDL interfaces to be integrated into one application, despite of their different implementations and running environment requirements.

In this paper we will present an approach for developing distributed component-based real-time software and the associated tools for cross-platform and cross-languages integration of real-time software components in distributed object environments. In Sections 2 to 4, we will describe our approach, including a distributed component architecture, the integration process and the central role of an integration tool. In Section 5, we will discuss some implementation issues. In Section 6, an example of a real-time airline reservation system will be used to illustrate our approach.

2. Our Approach

In a distributed system, it is common that different hardware systems running on different operating systems over different networks. To use the resources in these environments, software developers may choose different programming languages to implement the client/server or group software pieces. In order to make these software pieces to have the capability and flexibility to interact with each other, they should be developed by following a common distributed component architecture which specifies unified interfaces for method invocation, event handling, fault-tolerance and real-time services.

In our approach, besides functional requirements, a distributed component implements functions for publishing its interfaces to let other components know how to interact with this component through a Distributed Component Common Interface, for supporting group communication to satisfy fault-tolerance requirements through a Component Replication Interface, and for specifying real-time constraints to its services through a Real-time Specification Interface. All the interfaces are defined using interface definition language (IDL) of a common object environment, like CORBA.

Distributed components can be developed by wrapping these interfaces to the legacy software pieces. We can integrate these components into application software using an integration tool at a late stage. This tool allows the distributed application developers to adjust the components to offer preferable services and then connect the input and output events and invocations to each other through automatically generated component adapters. To satisfy fault-tolerance and real-time requirements, the integration tool also generates glue-code to maintain component groups and to monitor/schedule real-time invocations. Figure 1 shows our distributed component-based software architecture composed of the following three basic types of distributed components:

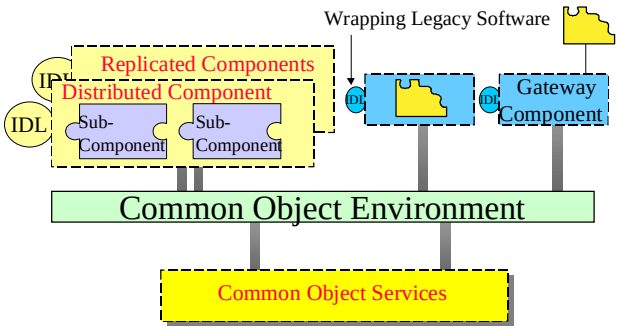

Figure 1: Distributed component-based software architecture

- **Regular Components,** which implement the basic functions defined in the distributed component standard interface in the Common Object Environment.
- **Group Components,** which are redundant copies of a component. They work together to offer consistent and continuous services.
- **Gateway Components,** which act like a gateway to a legacy software system. They translate requests that come from the common object environment into legacy requests and relay results out to the common object environment.

3. Distributed Component Architecture (DCA)

A distributed software component is a software piece which offers a set of services over a distributed system through its self-describing interfaces that are defined in a common representation language and its interface for reconfiguring some of the properties at

assembly time to fit into a specific application. In our distributed component architecture, we use CORBA as the integration bus because it offers a consistent distributed programming and run-time environment over most popular programming languages, operating systems and networks. Its Interface Definition Language (IDL) is suitable for specifying the component interface without the implementation details. A component can be programmed in a variety of languages on different operating systems (which now includes almost all the popular programming languages and operating systems). If a component implements a common set of IDL interfaces, it can interact with any other components through the common object environment daemon running on its native platform.

Each distributed component can be considered as an individual software part that implements component's common functions. All these common functions are defined in three related interfaces that together form a distributed real-time component architecture as shown in Figure 2.

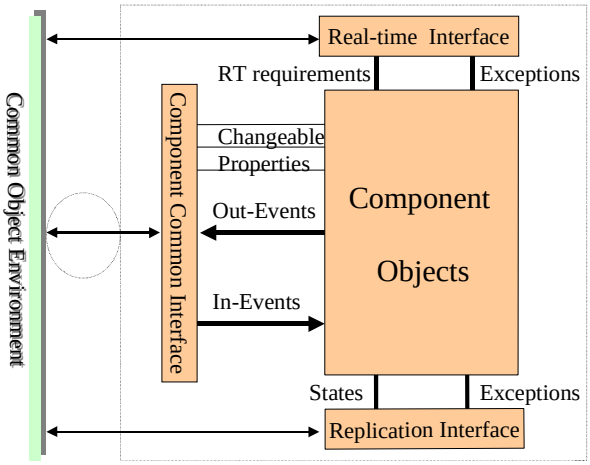

Figure 2: A distributed real-time component architecture

In the following, we will explain the three basic interfaces for a distributed real-time component:

3.1 Distributed component common interfaces

In order to offer a unified method for integration between distributed components with IDL language, a Distributed Component Common Interface is defined using IDL. To support distributed integration, this common interface includes the following four types of services:

- **Interface publishing and discovery**

Every component is a functional unit that offers a set of services through its public methods and data structure interfaces. In the component common interface, we define public methods which can return service names and their invocation interfaces upon being retrieved by an integration tool at the time of distributed component composition. To find a proper component that is offering the needed services, only syntax compatibility is not enough. It happens all the time that two methods or events with similar names are doing very different tasks. So additional semantic description about these public services should be part of retrievable information for a distributed component.

- **Event handling**

Every component receives incoming events as its input and generates outgoing events as its response. Besides these messages initiated by the components, there are events generated by the systems like mouse movements and exceptions. By defining a common data structure for events, the component can use the same method to accept different events and find the corresponding handlers. This interface allows the related components to react to remote events as well as local events when they are happening on different machines.

- **Persistent state maintenance**

During the integration and execution, some of the public properties of a component can be modified by the software developer or remote events. Persistent state maintenance interface allows the distributed components to store their states when the components become inactive and to restore the states before they are reactivated to offer consistent services through unified persistence functions.

- **Changeable properties support**

This interface allows some attributes of a component to be reassigned by the integration tool at the time of application composition from components so that the component can act according to the new values to fit better into the current application.

3.2 Component replication interface

In order to offer fault-tolerance feature in distributed component-based software, a component can implement additional functions to maintain a replication components group to enhance the availability and provide a certain degree of fault tolerance [15]. A critical service can be offered by several replicated components that reside in different hosts or processes. The replication interface will allow the replicated component group to initiate a global search-and-invoke when a request invokes their services. When some of the replicas that offer

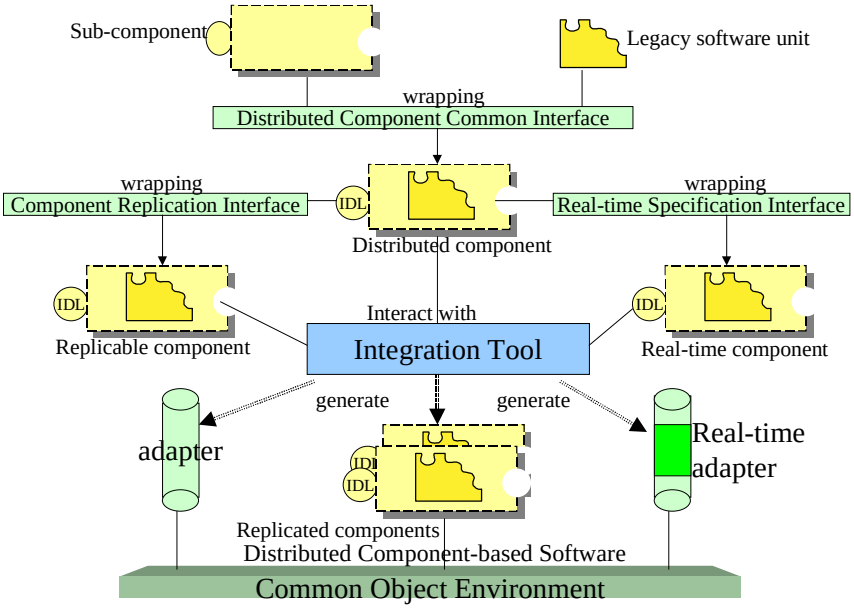

Figure 3: Our integration process of distributed components

the service are down, the others can still be found and respond to the request.

A key problem that needs be solved in distributed component replication is how to keep the state consistency among the replica [17]. The necessary functions that each replicable component needs to be implemented are defined in IDL format as Distributed Component Replication Interface, which includes the following two basic services:

- **Run-time component state retrieval and update**

These functions have the capability to combine component state variables into a predefined format (e.g. string stream) and the receiver of this state information could update its internal variables accordingly. All the replicas will interact through this interface to update or reconstruct their states.

- **Exception handling**

When one or several members of the replicated component group are down because of process crash or network connection being broken, the group should have the ability to renew their membership lists to exclude the mal-functional members. A client may connect to anyone of the member component by looking out the group name. When this component detects failure as it tries to communicate with other members, its exception generating method will invoke other members' exception handler methods for group states update.

3.3 Real-time specification interface

Real-time components have time-bounded requirements for remote method invocations. Inside a real-time component, the designer can carefully arrange the invocations and use effective techniques like multithreading and zero-buffering [18] to reduce the delays [19]. But, the major delays are usually caused by the outside communication like events transferring from component to component on different hosts.

To satisfy the real-time requirement, a run-time system with real-time invocation monitor and scheduler is required as a supplementary layer by interacting with the component through the real-time specification interface. This interface includes two main services:

- **Real-time requirements specification**

These functions specify the processing time ranges for the member methods that handle incoming events and the time bounds for outgoing remote method invocations of a real-time component. These requirements could be dynamically modified at the run-time according to previous processing or request history, or the change of states in the component if necessary.

- **Exception handling**

These functions can be called by the run-time monitors or schedulers as encountering exceptions like time bounds which cannot be satisfied during a remote method invocations.

4. Distributed Component Integration

The goal of defining distributed component interfaces is to support distributed real-time application software development through component integration. The integration process can be described in Figure 3. A distributed component can be developed by wrapping a legacy software unit

with the Distributed Component Common Interface. In addition, if the software unit also implements the Component Replication Interface, it has the capability to work as a group member. If it also implements the Real-time Specification Interface, it has the information and capability for real-time method invocations.

The next step is to compose distributed software from independently developed components. The integration is an interactive process of the user with an integration tool that understands these component interfaces and has the capability to 'glue' and 'tailor' these separate components into a system.

This component integration tool can interactively put the components together and to automatically generate skeleton and repetitive code for the components. Microsoft's Development Studio, Sun's Visual Studio and Borland's Delphi are such popular environments using visual programming, which allow programmers to visually compose application software using existing software components[16]. It enables rapid prototyping and rapid application development (RAD).

Traditional integration environments do not support distributed component integration. In our approach, we have developed a visual integration tool with transparent glue-code generation to facilitate integration and replication of distributed components.

Figure 4 describes the central role of the integration tool in real-time distributed application software development. Its basic functions are:

- **Visualize the distributed components and their interfaces**

  The integration tool allows the software developer to open the components that can be retrieved over the common object environment to construct a whole software structure using Drag-and-Drop.

- **Generate component adapter**

The integration tool generates an adapter for each component to link distributed methods using Distributed Component Common Interface based on user interaction. The link information is not available when a component is built. Only at integration stage, the developer specifies source and target methods of connections. The adapter stores this information for its component and establishes remote method invocations through the common object environment at run-time.

- **Modify properties**

The tool lists the public changeable properties of a component and allows the developer to modify their values so that a versatile component can be tailored to a specific application system.

- **Generate group adapters**

To achieve application-transparent state consistency among replicated distributed components, we use special group adapters with extra negotiation methods besides regular components. At the time of integration, the integration tool assigns the membership list and its prime membership ID into each group adapter's changeable attributes. Upon receiving remote requests, a group adapter forwards them to the prime member who will then broadcast to the others. All members of the group respond in the same way and therefore update their states accordingly while only the prime component will be allowed to send out remote requests through its group adapter if the current request triggers other components. If a member fails to contact

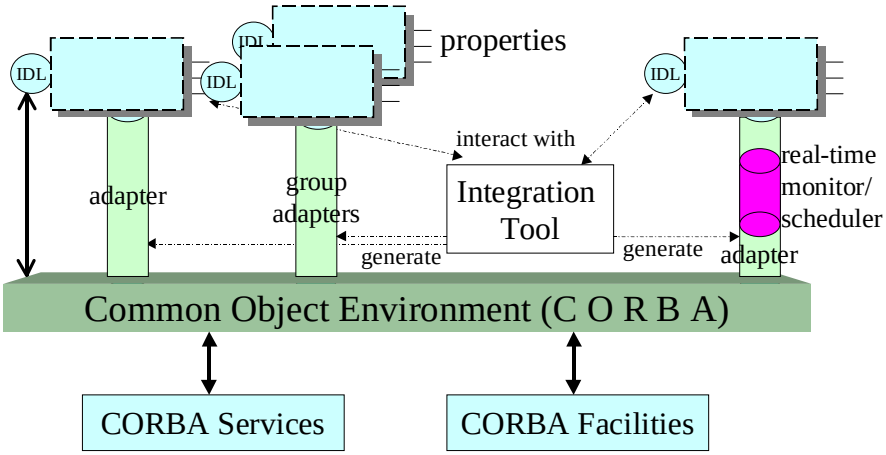

Figure 4: The role of the integration tool in constructing distributed component-based software

the prime host, it will broadcast exception to all the remaining members using Distributed Component Replication Interface, negotiate for a new prime replica, retrieve and re-establish state consistency when a new member joins as a new backup component.

- **Embed real-time request monitor and scheduler into the adapter**

For real-time components, their requests and responses need to be monitored and sorted [14]. The real-time request monitor is embedded on the out-going request component side. It keeps track of history response time and makes estimation on next response time. If it finds it is impossible to satisfy the time bounds requirement for a request, or after timing the possible request for close to the required time period, it will cancel the request and call the exception function of its component through Real-time Specification Interface. The real-time scheduler is embedded in components that accept incoming requests. It queues, sorts and releases method requests according to their emergency of the time limits to improve the response time. It also handles request cancellation for expired method invocation.

5. Implementation

We have developed a prototype of the distributed component interfaces, the integration tool and associated supporting adapter libraries based on IONA's Orbix 2.2 CORBA environment. This prototype is used to present and test the capability and efficiency of distributed component-based real-time application software development through cross-language (C, C++, Java) and cross-platform (Windows95, NT, Solaris) component integration.

5.1 Distributed component interfaces

The distributed component interfaces are defined in IDL 2.0. They can be translated into C/C++ on either Solaris or Windows 95/NT using Orbix 2.2's IDL compiler. We also use IONA's Java version CORBA --- OrbixWeb 3.0β to generate Java code from those IDL interfaces.

For interfaces publishing and discovering, every component registers its interfaces in CORBA's standard Interface Repository. By retrieving these interfaces from the multiple CORBA daemons, a component can discover compatible components that reside on different systems to interact with. To support accurate integration of components, instead of directly retrieving class information, we decide to have an Interface Descriptor which needs to be implemented for every distributed component. The interface descriptor includes a component's public method names which can be used to further retrieve signatures and the corresponding semantic description strings which can be prompted to the software developer to help him select correct components and methods in the system.

To handle remote events, a RemoteEvent class is defined as a base class for a component developer to define his own events, including names, properties and handlers. This allows the integration tool to retrieve event types and bind different outgoing events to the proper remote handlers using the same glue-code.

5.2 Integration tool

The integration tool, which is developed on both Windows 95/NT and Solaris, is a distributed software package based on Orbix. When it finds methods and events that defined in distributed components and displays them as icons under a component structure tree. Software developer can drag these icons to a working area and modify their public attributes and select a proper remote handler for the outgoing events of the components.

Because at the time of integration, the components are already in the mode of unchangeable executable files, it is improper to attach the adapter code directly to the component. We use a separate process on the same host to act as the adapter for each component so that the connection time between a component and its adapter will not be affected by network status which allows the real-time requests to be accurately monitored. Instead of generating all the glue-code at running time, the adapters, group adapters and real-time monitors/schedulers are pre-developed. At the time of integration, the tool specifies the connection source and destination to the adapter processes. Different adapter processes (regular adapter, group adapter, real-time adapter) are selected by the tool based on the type of the components (regular, replicable, real-time).

In a replicated component group, all the members' adapters share the same service name and only distinguish themselves by sub-names (like *marker* in Orbix [15]). As long as one of the member components is working, the client still can connect to the services through CORBA standard name binding service. When this component tries to broadcast the client requests to the others, it can detect the failure if some were down. It will send out exception events to the remaining members that will be handled through the exception handler functions to reconstruct the group.

When a component sends out a real-time request, its adapter will start a real-time monitor thread to time and record the response, and will generate exception events when the time is out. If a component offers real-time services to others, its adapter will incorporate real-time scheduler method which maintains a sorted request buffer according to the degree of emergency.

6. An Example

We will use a simple airline reservation system to demonstrate our approach. This system has several client consoles that can accept requests from end users and send requests to two travel agents that run on different hosts. If possible, the agents will offer the discount fares through a special booking approach with the airline. The airline system is running on several redundant servers to ensure its availability. The component architecture of the system is shown in Figure 5.

For this system, we define three kinds of distributed real-time components: Client, Agent, and Airlines. They are developed independently using different languages on different operating systems. The Distributed Component Common Interface has been implemented. All the public methods of the components are blocking invocations with return values.

♦ **Client**
Client component acts as the customer of this reservation system. It has two outgoing methods: MakeQuery (flight information), MakeReservation (ticket) and two changeable properties: aName (of client), aMaxWaitingTime (in seconds). The latter property specifies the real-time requirements for both outgoing methods. The Client component is developed using Java on Windows NT 4.0.

♦ **Agent**
This component acts as a travel agent. It has three incoming methods: AcceptQuery, Accept-Reservation, and AcceptDiscountReservation, and three out-going methods: MakeQuery, Make-Reservation, and MakeDiscountReservation. Its changeable properties are aName(of Agent) and aIsDiscount. If the latter property is true, the component is allowed to use method MakeDiscountReservation to connect to another agent's discount service. This component is developed using C++ on Windows NT 4.0.

♦ **Airlines**
Airlines component commits all the reservation and return flight information based on queries. It also implements Component Replication Interface that allows a group of Airlines components to work together as a fault tolerant software unit. It has two incoming methods: AcceptQuery, AcceptReservation and one changeable property: aName(of Airline). It is developed using C++ on Solaris 2.6.

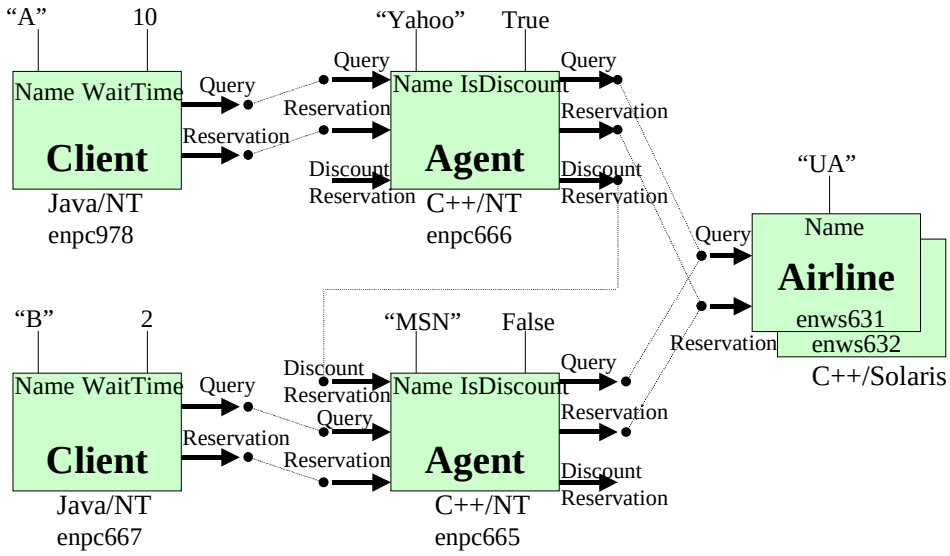

Figure 5: The component architecture of a simple Airline Reservation System

At the time of integration, two Clients components ("A" and "B") on two hosts enpc978 and enpc667, two Agents components ("Yahoo" and "MSN") on hosts enpc666 and enpc667, and two replicated Airlines components ("UA") on two workstations enws631 and enws632 are "glued" together through the automatically generated adapters which in this case all include real-time monitor/scheduler.

The integration tool retrieves all the public methods and changeable attributes from the Interface Repository on the remote hosts and generates connection adapters to relay remote events or exceptions to the components and assigns the connection adapters to each component through the predefined methods in Distributed Component Common Interface. The developer can also modify some attributes according to the application design (like aName, aIsDiscount, etc.). In this example, Client A has larger invocation time range than B, and hence its requests have lower priority than those from B in the real-time scheduler of Agent component. Agent Yahoo is allowed to make discount reservation through Agent MSN's AcceptDiscountReservation service because its property aIsDiscount is set to be true by the application developer. The two replicated Airlines components reside on different machines and share the same server name "UA" to form a fault tolerant service group.

7. Discussions

In the paper, we have presented an approach to distributed component-based real-time application software development. We have defined a distributed component common interface using CORBA's IDL. In order to have fault-tolerance feature in component-based software, a component replication mechanism was also developed. A real-time specification interface allows the run-time method invocations and offers timing-out exceptions handling through real-time monitor/scheduler. Since the monitor/scheduler also runs as a process in the local operating systems, the operating systems' real-time process/thread scheduling is required for satisfying hard real-time requirements. To implement this integration method, we have developed a CORBA-based integration tool with transparent glue-code generation to facilitate integration and replication of distributed real-time components. This distributed real-time component architecture divides the distributed application developing effort to two parts: component building and application integration which reduces the design and programming effort and improves the productivity of software developers and quality of software products.

References

[1] O. Nierstrasz, S. Gibbs and D. Tsichritzis, "Component-Oriented Software Development", *Comm. ACM*, Vol.35 No. 9, September 1992, pp.160-164.

[2] S. S. Yau, D. H. Bae and K. Yeom, "Object-oriented Development of Architecture Transparent Software for Distributed Parallel Systems", *Jour. Computer Communication*, Vol.16 No.5, May 1993, pp. 317-327.

[3] R. M. Adler, "Emerging Standards for Component Software", *IEEE Computer*, Vol.28 No.3, March 1995, pp. 68-77.

[4] J. Montgomery, "Distributing Components", *Byte*, April 1997, pp. 93-98.

[5] R. N. Taylor, N. Medvidovic, D. L. Dubrow and etc, "A Component- and Message-Based Architectural Style for GUI Software", *IEEE Trans. on Software Engineering*, Vol.22 No.6, June 1996, pp.390-396.

[6] Sun Microsystem, JavaBeans Specification, Version 1.0, 1997.

[7] Microsoft Corporation, *The Component Object Model Specification*, 1995.

[8] Sun Microsystem, Component-Based Software with Java Beans and ActiveX, whitepaper, http://www.sun.com/javastation/whitepapers/javabeans/javabean_ ch2.html, 1996.

[9] D. Garlan, R. Allen, and J. Ockerbloom, "Why Reuse is So Hard", *IEEE Software*, November 1995, pp.17-26.

[10] ComponentWare Consortium, Realizing a Virtual Application Warehouse Using ComponentWare, CWC whitepaper, http://www.componentware.com/vaw_wp.htm, June 1995.

[11] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, 1995.

[12] J. Siegel, *CORBA Fundamentals and Programming*, Wiley Computer Publishing Group, 1996.

[13] D. A. Lamb, "IDL: Sharing Intermediate Representations", *ACM Trans. on Programming Languages and Systems*, Vol. 9 No. 3, 1987, pp.297-318.

[14] J. Fraga, J. M. Farines, O. Furtado and F. Siqueira, "A Programming Model for Real-

Time Applications in Open Distributed Systems", *Proc. Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, August 1995, pp. 104-111.

[15] ISIS Distributed Systems, IONA Technologies, *Orbix+Isis Programmer's Guide*, Version 1.1, March 1996.

[16] S. P. Reiss, "Connecting Tools Using Message Passing in the Field Environment", *IEEE Software*, pp. 57-66, Vol. 7 No. 4, July 1990, pp. 57-67.

[17] M. Maekawa, A. E. Oldeheoft, and R. R. Oldeheoft, *Operating Systems Advanced Concepts*, Benjamin/Cummings Publishing, 1987.

[18] D. C. Schmidt, A. Gokhale, T. H. Harrison and G. Parulkar, "A High-performance Endsystem Architecture for Real-time CORBA", *IEEE Communications*, Vol.14, No.2, February 1997, pp. 34-41.

[19] S. S. Yau and D. H. Bae, "Object-Oriented and Functional Software Design for Distributed Real-Time Computing Systems," *Jour. Computer Communications*, Vol. 17, No. 10, October 1994, pp. 691-698.