An Abstract Object-Oriented Platform Model for an ATM Switching System

Young Man Kim
Department of Computer Science, Kookmin University
861-1, Chongnung-Dong, Songbuk-Gu, Seoul, 136-702, Korea
e-mail: ymkim@kmu.kookmin.ac.kr

Boo-Geum Jung **Eun-Hyang Lee** **Dong-Sun Lim**
Switching & Transmission Technology Laboratory, ETRI
161-1 Kajong-Dong, Yusong-Gu, Taejon, 305-350, Korea
e-mail: bgjung@etri.re.kr, ehlee@etri.re.kr, dslim@etri.re.kr

Abstract

In this paper, we present an abstract object-oriented platform model suitable for the real-time distributed telecommunication system. The proposed platform is constructed upon the extended version of the real-time, distributed operating system, *SROS*(Scalable Real-time Operating System)[6], that is developed at ETRI and successfully operated in the ATM switching system for several years.

The object-oriented software development and maintenance methodology will resolve the current software crisis in the area of telecommunication and switching systems due to the everlasting maintenance about the huge amount of the existing software and the ever increasing needs for the better and new communication services. In general, an object-oriented software platform realizes the object-oriented methodology and possesses many good aspects like high productivity, better reusability, easy maintenance, et al. The platform is also designed to present the distributed multimedia service in addition to real-time event delivery.

Recently, we have been implementing a couple of prototypes based on the proposed platform. Reflecting the evaluation results from these prototypes, the final platform specification will be determined.

Index Terms: communication software development, real-time object-oriented software platform design, distributed multimedia service, ATM switching system.

1 Introduction

Recent rapid evolution due to the technological and conceptual innovations in the communication and switching area has introduced software crisis again. The amount of software implemented for network/terminal control and service has become too enormous to be upgraded and maintained using conventional methodology. In particular, the new spectrum of network/switching technologies and services like real-time distributed multimedia service and mobile computing invited the need to a new generation of software development and the modification of the existing software modules.

To meet this challenge successfully, a new software development paradigm enhancing software productivity and maintainability much more than the conventional one is required in addition to the performance improvement of software development environment. Currently, the most promising methodology for leveling up software productivity and maintainability is known to be the object-oriented paradigm. The object-based modeling and design significantly improve a large software development process throughout all stages from the specification decision to the system implementation and maintenance. Furthermore, the components in the objected-oriented software consisting of a set of closely related objects can be reusable in the other applications without recompilation, producing software reusability.

Recently, the object-oriented distributed operating system, as a first category of object-oriented development environment, has been studied and prototyped in a large amount among the universities and the research institutes[1, 3, 4]. *Choices*[3] provides a framework for a customized operating system so that it can easily generate a particular version of OS by configuring the individual operating system object class templates. Thus, Choices not only consists of a set of objects, but also it provides object-oriented development interface. On the other hand, *Chorus/COOL*[1] offers the object-oriented software development and operation environment implemented in the COOL layer over micro-kernel, *nucleus* . *Spring*[4] modifies RPC mechanism to offer a client/server object invocation. In Spring object model, an object code itself is located together inside each client and the common underlying state of the object is shared by the server applications. Thus, the execution of object invocation is processed faster than that of conventional object model.

Another category of object-oriented environment is a middleware platform between an operating system and the application that provides a common object interface across networks[5, 8, 9]. As the representative middleware platform specifications are standardized, an application module is operative regardless of the kind of the particular implementation language used and the underlying operating system.

The most widely used middleware is CORBA[8] that has become a major platform in the object-oriented distributed platform market. Whereas, in the telecommunication and switching area, TINA[9] makes the general viewpoints of ISO Reference Model for Open Distributed Processing(RM-ODP)[5] concretely that are a set of guidelines for the design of object-oriented middleware. TINA architecture provides a set of concepts and principles to be applied in the specification, design, implementation, deployment, execution and operation of telecommunication software. In addition, a software architecture in TINA defines the constraints that should be applied during the entire software development process. Frequently, a TINA-based platform implementation is built upon CORBA since two specifications are close to each other. Thus, in some point of view, a TINA platform can be considered as a real-time extension of CORBA suitable for the particular need of telecommunication area.

CORBA utilizes Interface Description Language(IDL) for describing a set of methods of an object exported to the external object space, refered to as *object interface*. Although the inner implementation of CORBA is open and a vender uses its own proprietary structure and protocol, the interoperability between two different vender platforms is guaranteed via a common protocol called *GIOP*.

Although CORBA is a general-purpose object-oriented platform so flexible that it can be applicable to many major application areas, it can not be employed in the telecommunication and switching area due to the following reasons. First, any real-time concept does not exist in CORBA. There is no way to declare QoS and the other performance-related options about scheduling, channel demultiplexing, process dispatching, buffer management, etc. Second, CORBA is based on *request-reply* paradigm in the inter-object communication that is suitable to client-server model. On the other hand, in the telecommunication/switching system, another object communication paradigms are necessary for real-time object interaction, real-time event notification and multimedia data stream. To provide real-time property, each element composing the message processing and delivery in a platform should be adjusted to have controllability with optimized performance. Thus, to achieve this target, the underlying operating system and network modules should also support real-time characteristics.

In this paper, we present our approach to build an objected-oriented distributed platform that supports the real-time and multimedia application developments in the telecommunication and switching system. In the next section, an abstract object model for the real-time object-oriented platform is proposed. This model is materialized to the real-world object implementation model whose detailed descriptions are presented in [7]. In Section 3, the interfacing mechanism between an application program and the platform is explained with some illustrative examples. We conclude this paper in Section 4.

2 Abstract Object Model for Real-Time Object-Oriented Platform

In this section, we present an abstract object model[2] and object interrelation in object space in which computer resources like process, memory, network, node, etc., are invisible. An *object* is a minimum unit that provides some meaningful functions to the outside world through one or more *interface(s)* whose implementation details are unknown externally. Let's take an example of message object that is responsible for electronic message management. The manager object provides three interfaces: *message post* interface that offers a message repository service in behalf of the real destination object, *message notification* interface by which the manager object informs the destination object of a new message arrival, and *message management* interface that offers the configuration mechanism by which the exported services are configured. Fig. 1 depicts a message management object with three interfaces.

2.1 Communication Modes between Objects

Communications between objects in the telecommunication and switching systems are modeled into three modes. At first, *request/reply mode* models the communication pattern in a client/server application. A client object requests a server object to do some service by invoking the corresponding interface method exported by the server. After processing the request, the server replies with the results to the waiting client.

Second communication mode, *event mode*, represents an event notification message delivery for which a source object informs the destination object(s) of the occurrence of an event. In the telecommunication/switching system, such event should be delivered frequently in the real-time.

Third communication mode, *stream mode*, deals with multimedia data that occur periodically and in a real-time. Each time the voice/video data packet is produced, it should be delivered to the destination object in the fashion satisfying its QoS requirements, for example, maximum delay time, jitter, bandwidth, etc.

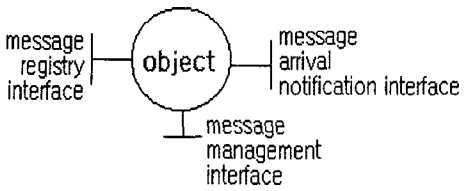

Figure 1: The manager object diagram

2.2 Object Binding

When a client object wants to use a service provided by a server object, the client should have an object pointer, refered to as *interface reference* (in short, *reference*), to invoke the corresponding interface method. The activity obtaining the reference of the server object interface is called *binding*. Binding mechanism is classified into two kinds according to the operation initiative: *implicit binding* and *explicit binding*.

In implicit binding, the communication medium bridging two objects are concealed from object space. On the contrary, in explicit binding, a *binding channel object* (in short, *binding object*) is exposed explicitly between the objects and all communications pass through the binding object. Furthermore, real-time and QoS requirements for the object communication in the telecommunication/switching system are managed by the binding object. Implicit binding is utilized in *request/reply* communication mode for the client/server application and explicit binding in event and stream mode for the real-time event and multimedia stream.

There are three types of implicit binding : (i) *static binding* for which the server is already prepared before the client initiates binding, (ii) *dynamic binding* for which the server is not existent yet in object space, and (iii) *indirect binding* for which the interface reference is transfered from another client that already did binding with the server. On the other hand, we can assort the explicit binding into three types according to the binding initiator: (i) *third-party binding*, (ii) *client binding*, and (iii) *server binding*.

In the following subsections, we will describe the binding operations in detail.

2.2.1 Implicit Binding

Static implicit binding occurs when a server object already registers itself to *object registry* before a client object tries to bind with the server. Object registry is a system object that manages all objects in object space. Fig. 2 dipicts the static binding process.

An object obtains the references of the system objects like object registry when it is created in object space. In the figure, server object also registers its interface(s) to object registry, for example, interface M (1). Later, when a client object wants to use the service provided by interface M , it inquires the interface reference to object registry (2) and then object registry replies with the reference (3). Finally, the client object invokes one or more services provided by interface M (4). Although a non-realtime object can be dynamically created and destroyed, it is desirable that a real-time object may be created statically and stay in object space permanently to omit the object creation overhead in the binding process time. Thus, binding with a real-time server object is typically static binding.

Fig. 3 depicts the dynamic implicit binding process. Object registry keeps track of the information about the object templates in addition to the active objects and its interfaces in object space. Using the object template, object registry can create a new object into object space. In the figure, a client object inquires the reference of interface M (1). As

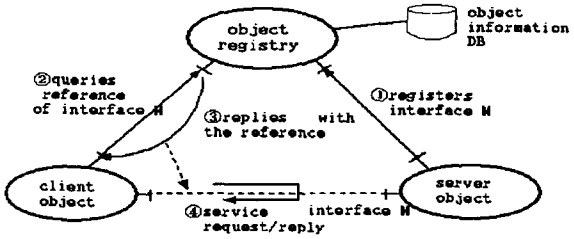

Figure 2: Static implicit binding process

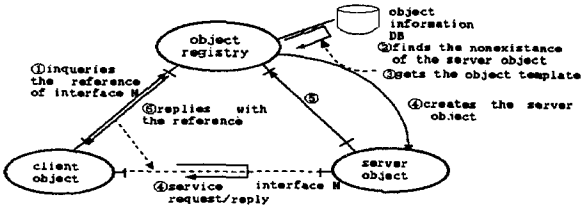

Figure 3: Dynamic implicit binding process

no corresponding server object exists currently, object registry creates a server object using the object template (2,3,4). When the server object finishes the initialization, it registers interface *M* into object registry (5). The remaining sequence is identical to that of static implicit binding.

Sometimes, a client object transfers the reference of some server object to another client when the latter explicitly requests it or the former wants to distribute its heavy workload to its fellows. Indirect implicit binding creats a new channel to the server object derived from the transfered reference. Indirect binding process is depicted in Fig. 4.

In the figure, the server object is already bound to client object *X*. When client object *Z* requests client *X* to inform the reference of interface *M* (1), client *X* decides the reference transfer to client *Z* for the purpose of the load balancing and replies with the reference of interface *M* (2). Then, client *Z* becomes connected to interface *M* and invokes some interface method(s) (3).

2.2.2 Explicit Binding

The real-time communication in the telecommunication/switching system is realized with a sequence of tandem communication components each of which should simultaneously satisfy the real-time/QoS requirement by allocating a set of available resources. To set up and manage the real-time/QoS property of the communication components explicitly, we introduce a *binding object* representing the communication components. A binding object is a communication object interconnecting two objects whose binding

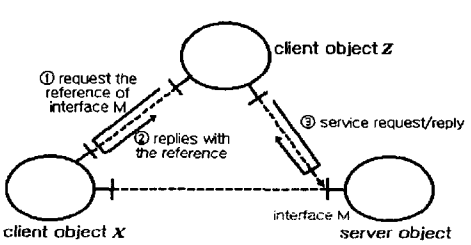

Figure 4: Indirect implicit binding process

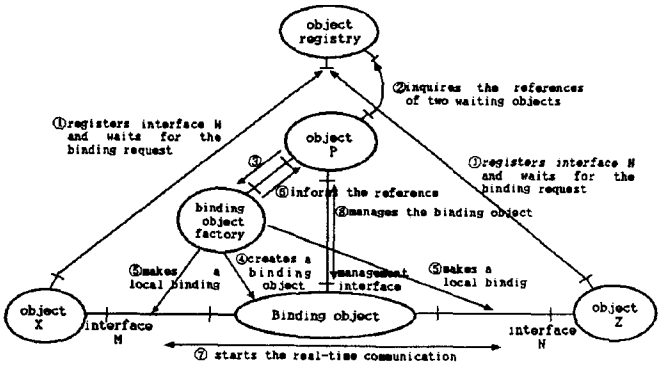

Figure 5: Third-party explicit binding process between two real-time objects

```
Interface <operation> CallManagement
{
   int callconnect () QoS (500, 2, 0, 0);
   void callreset ();
}option (PER_OBJECT, FIXED_PRIORITY, 10);
```

Figure 6: Interface *CallManagement* described in IDL

is done by *explicit binding*.

Before explaining the explicit binding process in detail, let us introduce another special object, *binding object factory*. A binding object factory is responsible for the creation of a particular binding object that bridges two objects and manages the real-time/QoS requirements. A responsible object can make an order to the binding factory to construct the corresponding binding object.

Explicit binding is classified into three types according to the binding initiator: *third-party binding*, *client binding*, and *server binding*. Fig. 5 shows the third-party binding process triggered by a third-party object *P* that binds two objects *X* and *Z*.

In the figure, objects *X* and *Z* want a real-time data transfer between them, e.g. multimedia packets or emergency status messages. At first, they register their own interfaces *M* and *N* and the corresponding real-time/QoS conditions (1). Later, third-party object *P* asks object registry the existence of the waiting objects (2). Since the references of objects *X* and *Z* are returned from the registry, object *P* makes a request to the binding object factory with the information about the references and the real-time/QoS requirements of objects *X* and *Z* (3). The factory examines the requirements to discriminate whether they can be realizable with the available system resources. If it reaches a positive conclusion, the factory creates a binding object (4) that has three references: two for the communication and one for the real-time/QoS management. Using the references, the factory makes the local bindings between the binding object and objects *X* and *Z* (5), and then sends the real-time/QoS management interface reference back to object *P* (6). Objects *X* and *Z* wake up and initiate the communication (7). On the other hand, object *P*, if necessary, dynamically controls real-time/QoS properties via the management interface of the binding object (8).

There are two variations to third-party explicit binding: client binding and server binding. In the former(latter) case, a client(server) takes charge of the role of the third-party object and initiates the binding process.

```
class CallManagement : public virual OpInterface
{
   public :
      static Call_Management * imbind (OpInterface *target);
      static Call_Management * exbind ();
      static Call_Management * exbind (Interface *target);
      virtual int callconnect();
      virtual void callreset();
};
```

Figure 7: C++ class declaration for interface *CallManagement*

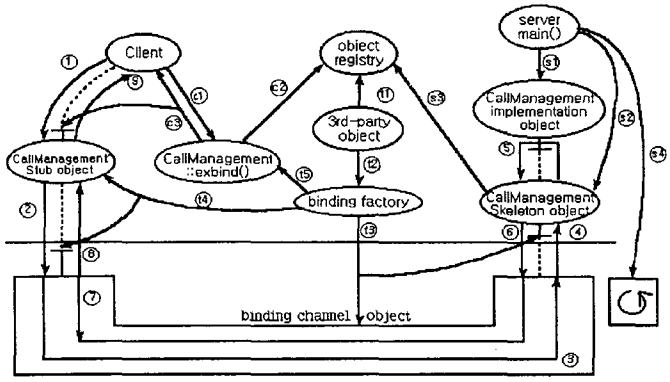

Figure 8: Third-party binding and the subsequent method invocation

3 Platform Interface

The object interface exported from an object is declared formally in an IDL interface file using Interface Description Language (IDL). For example, suppose that an application program is developed using C++ language although the other ones make no fundamental difference. The interface file is compiled to an object class declaration code, a stub code, and a skeleton code that are all written in C++. These auxiliary files are put together with a set of client/server source codes and compiled into a pair of executable client and server codes. Fig. 6 shows the IDL declaration of interface *CallManagement* exported from a call management server in the telecommunication/switching system.

In the figure, two methods are exported with two kinds of configuration options: QoS and real-time condition. First method *callconnect* is appended with a QoS condition demanding that the average frequency of the method invocation and its average response time should be 500 per second and 2 msec, respectively. Furthermore, according to the option at the end of interface statement, the object process that executes interface *CallManagement* should be allocated in per-object basis with fixed priority 10. When an object tries to bind to interface *CallManagement*, the corresponding binding factory will determine the realizablility of the QoS/real-time condition by examining the available resources in the system. If it can be satisfied, a binding channel is constructed, otherwise waiting in the binding request queue. The C++ class prototype corresponding to interface *CallManagement* is shown in Fig. 7.

First three functions in class *CallManagement* are used to create the binding channel objects according to implicit binding, third-party binding, and client binding, respectively. To explain the interaction between the application objects and the platform, let's follow a complete senario about third-party binding and the subsequent method invocation, as shown in Fig. 8.

First, the server component initializes itself by creating an object that implements interface *CallManagement* (s1), and its skeleton object (s2) that, in turn, registers itself to object registry (s3). Next, the server enters bind-waiting state (s4). Meanwhile, the client component calls method *exbind* offered by class *CallManagement* (c1) that registers the bind-related information necessary to construct a stub object (c2). Later, a third-party object asks object registry whether there are a pair of client/server objects waiting for binding to interface *CallManagement* (t1). If so, it calls the binding factory (t2) to create a binding object (t3), make a stub object in the client component and inform the stub of the local binding object reference (t4). Finally, the third-party object wakes up the client to start the invocation (t5). Then, the client returns from method *exbind* with the stub object reference (c3). From now on, the client can make any sequence of invocations to the server. When the client calls a method exported via interface *CallManagement* (1), the request message is routed to *CallManagement* implementation object (2, 3, 4, 5). After processing the request, the implementation object returns the result in reverse (6, 7, 8, 9). In this sequence, the stub/skeleton objects are in charge of marshaling/unmarshaling the parameter values and mediate between the binding channel object and the client/server objects, respectively.

4 Conclusion

In this paper, we proposed an abstract model of the real-time object-oriented distributed platform that is designed for the telecommunication/switching system. Currently, we are implementing some prototype platforms.

In the overall design process, we considered platform performance as the most important criterion to select the particular binding invocation mechanisms so as to minimize the response time and the message delivery overhead. The design and implementation of the prototype platform maximally utilizes the optimal mechanism provided by the real-time distributed operating system of ETRI, SROS[6], operating inside the ATM switching system successfully for several years.

References

[1] P. Amaral, R.Lea, and C. Jacquemot, "COOL-2 : An Object oriented support platform built above the Chorus Microkernel", Proc. Int. Workshop on Object Orientation in Operating System, Palo Alto, Oct. 17-18, 1991, pp. 68-73.

[2] G. Blair and J. B. Stefani, "Open Distributed Processing and Multimedia", Harlow, England, Addison-Wesley Longman Ltd, 1997.

[3] R. H. Campbell, N. Islam, and P. Madany, "Choices, Frameworks and Refinement", Computing Systems vol.5, no.3, 1992.

[4] G. Hamilton and P. Kougiouris, "The Spring Nucleus : A Microkernel for Object", Proc. 1993 Summer USENIX Conference, pp.147-160, June 1993.

[5] CD 10746-1 — ITU Recommendation X.901, "Open Distributed Processing-Reference Model-Part 1: Overview".

[6] S.I.Jun, et al., "SROS: A Dynamically-Scalable Distributed Real-Time Operating System for ATM Switching Network", GLOBECOM'98, Sydney, Australia, 8-12 Nov. 1998, pp.2918-2923.

[7] Boo-Geum Jung, Young Man Kim, Eun-Hyang Lee and Dong-Sun Lim, "Object-Oriented Platform Design for an ATM Switching System", ICC'2000, New Orleans, Louisiana, June 18-22, 2000.

[8] CORBA, "The Common Object Request Broker : Architecture and Specification(Release 2.0)", The Object Management Group, Framington, MA 01701-4568, USA.

[9] TINA-C, "Overall Concepts and Principles of TINA", TINA Baseline, TB_MDC. 018_1.0_94, Feb. 1995