An ATM Switching Software Using CTRON with Distributed Processing Support Function

Mitsunori Fukazawa, Yoshihiro Watanabe, Mituaki Kakemizu, and Moo Wan Kim
Fujitsu Laboratories Ltd.
1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan

Abstract
This paper describes the basic idea for ATM switching software using CTRON which has three-layer, object-oriented software architecture with distributed processing support function, and describes how to implement the architecture.

1. Introduction
Many organizations have researched ATM technology for use with B-ISDN. Various types of data such as high-speed data, low-speed data, constant-bit-rate data, variable-bit-rate data can be transmitted over the network by using ATM technology. This may provide us with communication services such as high-speed data communication, video broadcasting, and multimedia communication. ATM switching software will be needed to develop these services.

Traditionally, switching software has been designed to handle multi-processing and real-time-processing with high reliability. However, when requirements to the system like service enhancement have become greater, problems about productivity, maintainability and flexibility of the switching software have gradually been raised. These problems make software development and maintenance cost high and also cause a delay in service deployment. In the B-ISDN age, switching software will be required to handle more complicated processes and conditions.

Object-oriented software and the layered software architecture approach is regarded as one of the best methods to improve switching software[1]. Distributed processing is also effective.

This paper describes the basic idea for ATM switching software using CTRON[2] which has three-layer, object-oriented software architecture with distributed processing, and describes how to implement the architecture.

In Section 2, we describe the requirement of ATM switching software and in Section 3, we describe the basic idea for the software architecture and how to implement the architecture. In Section 4, we describe a prototype system.

2. Requirements
The main requirements of ATM switching software are as follows:

(1) Productivity
Customers using an ATM network will require many new services which handle various media and connection control such as multi-connection control and multi-party control. In addition, incorporation of B-ISDN and an intelligent network(IN) will accelerate the requirements for a large number of new services[3],[4].

The ATM switching software to implement these services will be much more complicated compared to the switching software used for a single media and a single connection control.

Service providers will have to develop new services rapidly to meet customer requirements. High productivity is important.

(2) Maintainability
The switching software architecture should be investigated to reduce the maintenance cost.

Although many new concepts and technologies are introduced into the ATM switching system, some of them will be changed due to unexpected factors. Therefore, ATM switching software should be designed so as to be modified easily and so as to localize the portion to be modified as small as possible.

(3) Flexibility
Generally, communication systems have very long life and ATM system will be introduced to the network in stepwise manner. The actual communication network is expected to consist of several sub-networks. Equipment for the network is provided by several different vendors each with their own design.

ATM switching software should be designed with
flexibility so that portability and inter-operability in a multi-vender and multi-OS environment is possible.

3. Basic idea of software architecture
We propose a three-layer software architecture based on CTRON, which uses an object-oriented software paradigm and a distributed processing support function for a distributed system environment. The basic idea is shown in Sections 3.1 to 3.3.

3.1 Layered software architecture
The basic idea of the layered software architecture is to group the ATM switching software into layers depending on its function. Each program must be designed to have the standard interface defined between layers. In this architecture, detailed information in a layer is hidden to other layers software. A software developer needs to know only about the interface. This feature assists in developing large-scale software like switching software. Additions and modifications are restricted to one layer at a time, thus making it easier to maintain the software. The software is also more portable.

3.1.1 Overview of each layer
Figure 1 shows a three-layer software architecture based on CTRON.

![Fig.1 Three-layer software architecture](image)

(1) Basic OS layer
The basic OS(BOS) layer consists of a kernel and I/O drivers which control the hardware. This layer provides the standard interface to the upper layer software. Therefore, the upper layer does not depend on the hardware architecture.

(2) Extended OS layer
The extended OS(EOS) layer manages the switching resources. The switching resources are controlled by instructions from the application layer. High-level interfaces are provided to the upper layer. This improves software portability and makes it easier to develop application programs.

(3) Application layer
The application layer contains call processing related software. Service specific parts are described in the software of this layer. Application layer software uses the EOS functions through the EOS interface. Because EOS functions don't depend on any specific service, service specific and service nonspecific parts are thus clearly separated. This makes it easier to add and change the software when a service specification is changed.

3.1.2 ATM switching resource control interface
Currently, CTRON defines a switching resource (such as switch fabric, circuit, etc.) control interface only for a narrow-band switching system. The interface for an ATM switching system is not defined clearly in the present CTRON, and more detailed interface must be defined. In the following, we describe how to define the ATM switching resource control interface between layers.

Several ATM switch and traffic control methods are being proposed now, and the ones to be used have not yet been decided. Those functions will be added or modified in the future. How switch control functions should be layered is a key point in designing ATM switching software which has high maintainability.

The interface to control ATM switching resources is defined based on ATM switching resource control functions in the layer and the ATM switching resource view seen from the upper layer.

There are three abstraction levels of such views corresponding to the three layers. Figure 2 shows the functions provided by each layer and the relationship between the view and the layer.

![Fig.2 ATM switching resource view](image)
view in the right part of the figure corresponding to each of the two layers. On the other hand, there is no logical model of the switching resource for the BOS layer.

1) Basic OS Interface
Several new ATM switch architectures such as common-buffer, output, and input-buffer architecture have been proposed. Control function of an ATM switch fabric developed based on ATM specific technology is provided by the switch driver located in the BOS layer. The switch driver should hide the difference of the ATM switch fabric architecture from the upper layer so that the upper layer software may have portability.

The BOS interface, which is able to assign and connect the speech path only by specifying the highway number, virtual path identifier(VPI), and virtual channel identifier(VCI) of the input line and output line, is defined.

The BOS layer has many other drivers than the switch driver. BOS prepares the IOCS(I/O Control System) interface for the EOS layer software. IOCS provides common interface which does not depend on any driver. This contributes to the high productivity and maintainability of the EOS software.

2) Extended OS Interface
It is assumed that the switching service program described in the application layer is to assign the logical switching resources for the service requirements of the user. In this layer, mapping between the two logical models described above is done. Functions which treat VPIs and VCIs are located in this layer. For example, the call admission control function is necessary to assign the ATM switching resources to a call. The outgoing VPI selection function for an ATM switching system which manages the bandwidth of each outgoing VPI is also needed to allocate a outgoing VCI. A simple interface should be provided to the application layer which only returns whether the requirements such as a logical subscriber line assignment are acceptable.

3.2 Object-oriented software
Concerning the conventional procedure-oriented software paradigm, it is difficult to predict how much procedures are influenced by the modification of some data structure or procedure, because many procedures can access the same data. This makes it difficult to maintain switching software. An object-oriented software paradigm is effective in solving this problem.

In the object-oriented software paradigm, data and procedure are encapsulated into an object. Data access among objects is limited by the message interface so as not to be able to access other object's data directly. Each object is thus highly independent from others and is not effected by modification of other objects. The ATM switching software is designed to have high maintainability by adopting the object-oriented software paradigm. An object is regarded as reusable software parts because of its high independency. The inheritance mechanism of the object clarifies the software structure and helps reduce the program size. These features increase productivity and maintainability.

3.2.1 Object Configuration of the ATM switching software
Figure 3 shows the object configuration of the ATM switching software related to service execution.
(1) BOS layer objects
The BOS layer consists of a kernel, driver objects, and IOCS. Switch driver object manages bandwidth of the internal speech path of the ATM switch fabric. And it allocates, connects and disconnects the internal path. The kernel and IOCS provide program execution environment and essentially independent from other software. Therefore these are not necessarily designed by the object-oriented paradigm.

(2) EOS layer objects
The EOS layer is divided into two parts. One of them consists of objects which control the logical ATM switching resource. The other consists of software which provides a distributed processing support environment. Each switch control object corresponds to an element of the logical model of the ATM switching resources described in Section 3.1.2. The subscriber line and trunk line management objects manage the vacant status of a VCI and the usable bandwidth of a VPI. Acceptance of a new call is determined by a call admission control object with usable VPI bandwidth managed in each line management object. A path control object hunts a speech path between the logical subscriber and trunk lines in the logical switch fabric. This object also connects and disconnects that path. Subscriber and ISUP signal control objects execute signaling protocol processes for logical subscriber, and trunk lines. gCTRON and RPC realize the distributed processing support environment and these are not necessarily designed by object-oriented paradigm for the same reason as the kernel and IOCS.

(3) Application layer objects
The application layer consists of call processing related software. There are two methods to realize the call processing function as an object. One of those is to separate a call into two part, the originating and terminating part, and objects which control each part are implemented. The other method is to implement a single object for a call, which integrates the control of the originating and terminating part. In the first method, there is overhead with communication between two objects(i.e. originating and terminating control objects) during the call processing. However, this method enables objects to manage fewer number of call states than the second method. As described in Section 2, the complexity of the service logic will be one of the essential problem of the switching software. Therefore, we adopt the first method which reduces the complexity. Caller and callee objects are the software entity which control the originating and terminating part of a call. A service specific process is implemented by the service scenario object. Other objects in this layer include various analysis objects such as subscriber and service analysis.

3.2.2 Implementation of object-oriented programming
The object-oriented software paradigm is an effective solution to meet the requirements explained in Section 2. However, depending on the implementation method, performance of the software is lower than that of conventional software. Efforts should be made to decrease the overhead of object-oriented programming to as little as possible. Implementation methods of object-oriented programming such as allocation of the instance variable area and message communication which affect the performance are discussed.

(1) Basic design of object implementation
Objects are implemented using the structure and function of the C language taking into account the performance aspects. The structure consists of instance data area and method pointer area as shown in Figure 4. Each method pointer area has a pointer to the memory area where the corresponding method is stored. The inheritance of methods is realized by having the same value of the method pointer of super-class and sub-class. We don't use the dynamic binding of a method and procedure using method search algorithm because it generates to much overhead.

(2) Basic design of object execution and message communication
An object is assumed to be a self-completed function module. The method in which each object is executed on its own task communicating with each other asynchronously, is thus natural and comprehensive. However, this method requires many tasks to be created for a single call. The resources managed by an operating system are thus consumed much. This lowers multi-
processing performance. In addition, the task creation, task switching and asynchronous communication for sequential processes generate the overhead. This makes realtime-processing performance low. Another method is thus needed.

Figure 5 shows our approach. In the switching system, most of the processes in a single call are executed sequentially. Caller and callee objects controlling the call which is the unit for multi-processing is executed on their own task. However, service scenario objects, various analysis objects, and resource control objects which are under control of the call and executed sequentially, are invoked by function call and executed on the task of the caller or callee object. Subscriber and ISUP signal control objects which execute signal reception processes asynchronously also have their own tasks. As a result, message communication between objects being executed on the same task is achieved by function call. Message communication between objects being executed on different tasks is achieved by asynchronous communication using a message box.

(a) First method: Static allocation
The maximum number of simultaneous generation of instances is forecast. The area for instance objects is allocated beforehand as an array at a fixed address. The empty instance area is controlled by the busy-idle bitmap table. When the instance generation is requested, the bitmap table is examined and the beginning address of the empty area is returned.

(b) Second Method: Dynamic allocation
This method is adopted by Smalltalk-80. An instance object is generated by transmitting the message named "new" to the class objects, and the instance variable area is allocated dynamically. The number of processing steps for dynamic memory allocation is different depending on the memory management system. However, it obviously requires more steps than static allocation. Static allocation has the following advantages:
- number of dynamic steps is small
- memory usage efficiency is high if call data size is not different for each call
- call relief process is easy to achieve

Dynamic allocation generates several dynamic steps, but it enables users to use memory efficiently, being independent from the required data size allocation. When B-ISDN is first implemented, the static allocation method will be sufficient because services will be simple and variation will not be so large. However, when the variation of the services grows as described in Section 1, the call data size required will differ depending on the service. A combined method of the first and second methods or a new method which balances the dynamic steps and memory usage efficiency will thus be needed in the future.

3.3 Distributed processing support environment
To provide the heterogeneous system described in Section 2 to meet the customer's requirements, the distributed module switching system shown in Figure 6 is effective.

There are several kinds of switching modules such as those for ATM, N-ISDN, analog, etc. These modules are added and removed easily in a building block manner. This architecture enables network operators to integrate OA&M activities and handle such activities easily. However, there is a new communication pattern (inter module communication) to be handled in this architecture. This is a problem which makes the switching software more complex than conventional
software. There are two major approaches to resolve this problem. The first approach is to regard an intra-module communication as an inter-module communication virtually. The second approach is to construct the distributed processing support environment. We adopted the second approach because this environment gives application software high flexibility and has enough capability to accommodate other problems involving several modules or nodes[5],[6].

Fig. 6 Distributed module switching system

3.3.1 Basic idea of gCTRON

A distributed processing support environment requires the functions described below [7].
- Access transparency
- Location transparency
- Fault isolation and tolerance
- Distributed resource management
- Heterogeneity isolation

To achieve these functions, the distributed processing support function, which we call gCTRON (global CTRON), is implemented on top of CTRON. Figure 7 shows the basic idea of gCTRON; that is, to make a collection of processors in a loosely-coupled system appear as a single machine to the applications. When running over gCTRON, applications on different processors can execute many of the normal CTRON functions. The special feature of gCTRON is that it allows these functions to be performed in a global sense. By making gCTRON system calls, applications can access objects1 on other processors in a transparent manner. The objects manipulated by gCTRON are much the same as those handled by regular CTRON. In gCTRON these objects can exist anywhere in the system, and can be accessed from anywhere in the system. To distinguish them from CTRON objects, we refer to gCTRON objects as global objects. Because of this feature, call control application software must not be developed independently for the inter-module and intra-module communication services. The same application software can be used for both services by using gCTRON.

Fig. 7 gCTRON model

3.3.2 Implementation of gCTRON

(1) Design Issues

Naming and security are major design issues to be considered. These are handled together in the distributed processing support environment provided by gCTRON. First, we describe these issues briefly.

(a) Naming

In a distributed system, logical names are used by applications to refer to resources located throughout the network. A gCTRON manages the name space and maps logical names into physical identifiers.

(b) Security

Security, i.e., secrecy and privacy, must be considered, when software is modified or new modules are added. Interactions between modules must not cause any faults, and security from unauthorized users must be maintained.

(2) Capabilities

The core security and naming system is based on the notion of capabilities[8]. Capabilities are globally unique, dynamically created tokens which serve to identify a global object. Access to an object is restricted to the owner(s) of its capability. Capabilities can be freely passed among applications to share system resources. To ensure security, capabilities are encrypted. Application programs use gCTRON services by means of system calls. In gCTRON system calls, applications reference an object by passing a pointer to the capability. Capabilities are implemented as a data structure. This is shown in Figure 8.

1. The term "objects" in this case refers to software resources as defined by the operating system specification, e.g., tasks, semaphores and message boxes. These should not be confused with programmatical objects.
The host-id field specifies the host CPU on which the object resides. The obj-type field describes the type of global object associated with the capability. gCTRON uses the services of CTRON to create global objects. The system tag returned by CTRON at the time of creation is used by gCTRON as the obj-id field in the capability. When an object is created, gCTRON also generates a random number and includes it in the random field. The random field is used to discriminate between old objects which have been deleted and new objects which reuse the obj-id(system tag) of the old object. The host-id, obj-id, and random fields make up a key to identify a global object in the distributed system.

The rights field is a bit-map describing the ways in which a capability holder may access an object. For example, rights field for a message box has bits representing the rights to delete, sends to, remove, and get information about the message box status. The final field, check, provides security in the capability system. The check field is generated by computing a checksum over the preceding five fields, then encrypting the result with a randomly generated key. The key is held securely in a table within the gCTRON. When a capability is presented to gCTRON as part of a request, it is validated by generating a checksum over the submitted capability's first five fields and comparing the result with the submitted check field.

(3) Public Names
Since capabilities are dynamically generated, communication deadlock problem occurs in some case. Figure 9 illustrates this problem.

Using a gCTRON system call for creation of a global message box, application instance APL1 on CPU A creates a global message box MB1 on CPU Z which it will use to accept messages. In a similar fashion, application instance APL2 on CPU B creates a global message box MB2 on CPU Y. APL2 would like to send a message to APL1 via MB1 but it does not know MB1's capability. Furthermore, APL1 would like to inform APL2 of MB1's capability but it doesn't know MB2's capability. A way to resolve this deadlock problem is to provide a static naming mechanism. This is what the public naming system does. The method of public naming is illustrated in Figure 10.

When an message box is created under gCTRON, the user application may specify a name to be bound to the message box. The name-to-capability binding is recorded in a table by a name server. Any application instance in the distributed system which knows the name can learn the dynamically generated capability associated with it by using a gCTRON system call for getting capability based on a public name. Then it can use the capability to send a message to the message box. It is assumed that both APL1 and APL2 know the names of MB1 and MB2, but not necessarily the capabilities. The specification of a name when creating an object is completely optional in gCTRON. If a name is specified, it must be globally unique.

(4) Communication
Besides naming and security functions, the gCTRON
contains primitives for handling interprocessor communication.

Inter-processor communication in gCTRON is based on the Remote Procedure Call (RPC) paradigm. In this paradigm, application instances on a local processor can make calls to procedures on remote processors. This allows both local and remote procedure calls to be used. An RPC runtime system takes care of the inter-processor communication details.

At first glance, it may seem that message-based inter-processor communication is the better choice, since the CTRON specification uses a message-based paradigm. However, when acknowledgments are included and one explores the possibilities of marshaling data into and out of messages, the whole mechanism begins to look very similar to RPC. In addition, it is straightforward to create a message-based interface to the communication system by adding a layer on top of RPC.

4. Prototype system
4.1 Hardware configuration
Figure 11 shows the hardware configuration of our experimental system.

The ATM switch uses a multi-stage self-routing switch developed by Fujitsu[9]. Switch throughput is about 600 Mbps. The G-micro control system processor is based on TRON installation specifications. The performance of this processor is about 17 Mips. At the present time, inter-processor communication is done via ethernet for the time being. Cell assembly/disassembly (CLAD) converts between packets and cells. The signaling facility (SIG) processes the second layer of subscriber signaling.

4.2 Software configuration
The BOS consists of a kernel, drivers and an IOCS as shown in Figure 3. We implemented the switch driver object. The multi stage switching architecture is hidden from the upper layer software by this object. In the EOS layer and the application layer, we implemented all software except subscriber and ISUP signal control objects.

These software except gCTRON and RPC is implemented based on the idea of object-oriented programing implementation discussed in Section 3.2. The other software in Figure 3 already developed for N-ISDN switching system and some of them were enhanced for the ATM switching system.

Currently, only point-to-point connection service is implemented as service scenario objects. The size of the application, switch control and switch driver objects are about 7500, 2000, 2500 lines of C code(these software doesn't include exception handling and restart process.)

4.3 Distributed Processing Environment
The following collection of six system calls for gCTRON has been implemented.
1. gCRE_MBXO: global message box creation.
2. gDEL_MBXO: global message box deletion.
3. gREF_MBXO: get status information about a global message box.
4. gSND_MSG(): send a message to a global message box.
5. gREM_MSG(): retrieve a message from a global message box.
6. gGET_CAP(): look up a capability based on a public name.

The full gCTRON interface will eventually include a total of about 31 system calls.

The current implementation is composed of about 5000 lines of C code, and uses approximately 50K bytes of memory. Another 50K bytes of memory is needed for data tables.

5. Conclusion
We proposed ATM switching software using CTRON which is based on a three-layer, object-oriented architecture with a distributed processing support function. In the three-layer software architecture, we proposed interfaces between the basic OS and extended OS and between the extended OS and applications for
the ATM switching system. In the object-oriented software architecture, we proposed how to allocate an instance variable area and how to communicate among objects. Then, we proposed a distributed processing support function which supports communication among the application objects using RPC. Finally, we described our prototype system based on these software architectures.

Acknowledgements
The authors would like to thank Dr. H. Takanashi and Dr. K. Murano for arranging this research and also thank Mr. G. D. Flinchbaugh of Bellcore for his collaboration research effort.

References