DOVE : 가상 계산 환경을 위한 분산 객체 시스템

(DOVE : A Distributed Object System for Virtual Computing Environment)

김 형 도 † 우 영 제 † 류 소 현 † 정 창 성 ††
(Hyeong-do Kim) (Young-je Woo) (So-hyun Ryu) (Chang-sung Jeong)

요 약 본 논문에서는 객체 지향 분산 가상 컴퓨팅 환경인 DOVE에 대하여 기술한다. DOVE는 독립적인 분산 객체들이 메소드 호출을 통하여 서로 상호 작용하는 분산 객체 모델을 기반으로 설계되었으며, 다수의 이기종 머신들로 구성된 분산 환경을 하나의 논리적인 단일 가상 컴퓨터로 사용자에게 제공함으로써 원격지에 있는 분산 객체들이 하나의 가상 컴퓨터에 존재하는 것처럼 사용할 수 있도록 한다. 또한, 병렬성, 이기종 환경, 객체 그룹, 단일한 네임 서비스, 그리고 오류 허용 등의 지원을 통하여 병렬 프로그램 개발을 위한 투명성 있고 사용이 용이한 프로그래밍 환경을 제공한다. 병렬성은 다양한 메소드 호출, 객체 그룹을 통한 다중 메소드 호출, 다중 쓰레드 구조 그리고 여러 동기화 구조를 사용함으로써 효과적으로 지원되며, 자동화된 데이타 변환 코드 생성, IDL 컴파일러를 통한 stub와 skeleton 객체 생성 그리고 객체 관리자를 통한 객체 라이프 관리와 네임 서비스를 통하여 이기종 간 호환성 문제를 해결하였으며 투명성 있고 사용이 용이한 프로그래밍 환경을 제공한다. 자치성 있는 분산 객체와 다중 레이어 구조 그리고 분산화된 네임 서비스와 객체 관리 구조를 사용함으로써 확장성과 보수성이 향상되었으며, 비동기 방식의 사건 및 예외 처리 통한 오류 탐지 및 확인 기능을 제공한다.

Abstract In this paper we present a Distributed Object oriented Virtual computing Environment, called DOVE which consists of autonomous distributed objects interacting with one another via method invocations based on a distributed object model. DOVE appears to a user logically as a single virtual computer for a set of heterogeneous hosts connected by a network as if objects in remote site reside in one virtual computer. By supporting efficient parallelism, heterogeneity, group communication, single global name service and fault-tolerance, it provides a transparent and easy-to-use programming environment for parallel applications. Efficient parallelism is supported by diverse remote method invocation, multiple method invocation for object group, multi-threaded architecture and synchronization schemes. Heterogeneity is achieved by automatic data marshalling and unmarshalling, and an easy-to-use and transparent programming environment is provided by stub and skeleton objects generated by DOVE IDL compiler, object life control and naming service of object manager. Autonomy of distributed objects, multi-layered architecture and decentralized approaches in hierarchical naming service and object management make DOVE more extensible and scalable. Also, fault tolerance is provided by fault detection in object using a timeout mechanism, and fault notification using asynchronous exception handling methods

† 비 회 원 : 고려대학교 전자공학과
kimhd@snoopy.korea.ac.kr
gossamor@snoopy.korea.ac.kr
messias@snoopy.korea.ac.kr
†† 종신회원 : 고려대학교 전자공학과 교수
csjeong@charlie.korea.ac.kr
논문접수 : 1998년 12월 4일
심사완료 : 2000년 2월 1일

1. Introduction

During the last decade, distributed computing has been a popular way to build applications for several reasons: performance, cost effectiveness, sharing of resources, fault tolerance, and etc. By connecting several machines together, we can access more computing power. Groups of workstations can provide

very high performance more cheaply than traditional supercomputers or parallel computers. Some resources like large databases or expensive hardware such as supercomputers cannot be affordable by every site. Distributed computing permits users in different locations to share expensive resources, thus leading to the economic benefits through better utilization of resources. In a distributed system users can be protected from hardware or software failures by replicating important services or programs at physically separate sites.

However, just connecting computers together is not sufficient. Distributed computing also bear inherent problems to be dealt with. The communication networks in a distributed system provide a low throughput and long delays compared to the links employed in multiprocessor parallel machines. Moreover, the data delivery in computer network is not ordered and unreliable. Therefore, distributed computing may result in unacceptable performance when applied on fine-grained parallel applications. Most modern operating systems support network capability, allowing us to do simple remote operations such as execute processes, or access files, but to build real distributed applications using these raw tools requires a lot of work. Naming systems for nodes and processes must be created. Communication protocols must be embedded in the application code. Implementation details at this level vary from machine to machine, even between different versions of an operating system. Therefore, we need a programming model for distributed computing which allows users to write a portable code, and hide the details of process control and communication from the programmer by supporting consistent interface for data exchange and treatment of failures on a heterogeneous environment. Without easy-to-use software programming tools or libraries, the network programming will be too complex for most users.

Recently, distributed object model such as OMG CORBA[1], JAVA/RMI[2] and DCOM[3] have been introduced to tackle the problems inherent in distributed computing on a heterogeneous environment. Distributed object model consists of objects which interacts via method invocation, while message passing model consists of processes which communicate with each other via message passing. Distributed object model has several benefits over message passing model. It provides an easy programming environment by supporting transparency of distributed objects, plug and play of software as well as the advantages of object oriented programming such as reusability, extensibility, and maintainability through abstraction, encapsulation and inheritance. However, it lacks some functionalities for parallel applications, since they are based on client-server model. It does not support group operations, and has some difficulties in implementing efficient parallelisms by using asynchronous communications. Further, they did not integrate core services with their models such as object creation, location, deletion and name service which are indispensable to provides single system view and distribution transparencies.

In this paper we present a Distributed Object oriented Virtual computing Environment, called DOVE which consists of autonomous distributed objects interacting with one another via method invocation based on distributed object model. It appears to a user logically as a single virtual computer for a set of heterogeneous hosts connected by a network as if objects in remote site reside in one virtual computer. By supporting efficient parallelism, heterogeneity, reliable group communication, single global name service and fault-tolerance, it provides a transparent and easy-to-use programming environment for parallel applications as well as distributed ones. Efficient parallelism is supported by diverse remote method invocation, multiple method invocation for object group, multi-threaded architecture and synchronization schemes. Heterogeneity is achieved by automatic data marshalling and unmarshalling, and an easy-to-use and transparent programming environment is provided by stub and skeleton objects generated by a CORBA compliant IDL compiler, object life control and naming service of the object manager. Autonomy of distributed objects, multi-layered architecture and decentralized approaches in

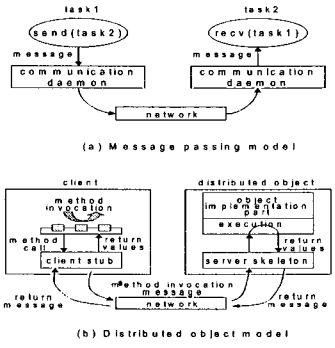
hierarchical naming service and object management make DOVE more extensible and scalable. Also, fault tolerance is provided by fault detection in object using a timeout mechanism, and fault notification using asynchronous exception handling methods. One object manager runs per host, and manages object life control for object creation, deletion and location, naming service and fault detection. These days the computer networks are becoming larger, faster, and the performance of machines more powerful. Therefore, DOVE is more attractive in that faster and more powerful, high performance machines and workstations over a local or wide-are networks can be connected to make a single virtual parallel computer with high computing power at low cost.

The rest of the paper is organized as follows: In section 2, we describe previous works which are related to our work. In section 3, we describe details of our object model as a base model of DOVE including remote method invocations, multiple method invocations and asynchronous event/exception handling. In section 4, we present DOVE run-time system. In section 5, we describe object management. In section 6 and 7 we present the implementation of DOVE and its performance evaluation respectively. Finally, a conclusion will be given in section 8.

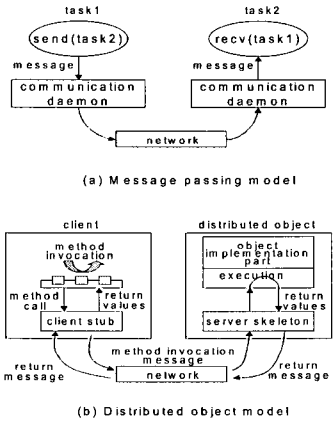
2. Related Work

In this section, we describe several distributed computing systems which are related to our work. Distributed programming model can be broadly classified into a message passing model and a distributed object model. (See figure 1.) In the message passing model, a program is divided into components or tasks, which may run on different nodes of machines. The tasks communicate with each other by explicitly sending and receiving messages. In the distributed object model, often called a method invocation model, a program consists of distributed objects which interact with one another by the method invocation. PVM[4,5] and MPI[6] are distributed programming systems based on the message passing model, while CORBA[1] and Legion [7] based on method invocation model.

MPI is a single standard programming interface mainly designed for developing high performance parallel applications with emphasis on a variety of communication pattern and communication topology. However, MPI lacks in functionalities such as process control, resource management and fault-tolerance. PVM is one of the most widely used distributed computing systems based on the message passing model, and connects together separate physical machines into a virtual computer by providing process control, simple message passing and packing constructs and dynamic process group management. In PVM, a daemon process, which runs on each host of a virtual machine, is used not only as a process



(a) Messaqe passingmodel



(b) Distributed object model

Fig. 2 Message passing model and distributed object model

controller but also a message router, which may result in communication bottleneck as all tasks heavily depend on daemon processes.

Legion is an architecture based on a distributed object model, and designed to build system service which provides a virtual machine using shared object, shared name space, fault-tolerance. Legion uses a data flow model as a parallel computation model, and parallelisms are implicitly supported by the underlying runtime system. However, the management of data dependency graphs for every invocation as well as scheduling nodes of the graphs may incur additional computation overhead, and no support for object group may cause communication inefficiency. CORBA is a vender-independent standard which aims at interoperability and portability of distributed applications. CORBA defines an distributed object model for accessing distributed objects. It includes an Interface Description Language, and a specification for the functionality of run-time systems that enable access to objects. But CORBA is based on client-server model rather than a parallel computing model, and hence it is not adequate to provide a virtual machine. Also, it is not well suited to distributed applications where performance requirements demand asynchronous communication and multiple method invocations.

3. DOVE Object Model

DOVE is based on distributed object model which consists of several distributed objects interacting with each other using method invocation mechanism. A distributed object is divided into an *interface object* and an *implementation object*. The interface object is distributed to applications which are to use the distributed object, and it provides interaction point to its implementation object. Users can issue a method invocation to a distributed object by invoking the method of its interface object, a local representative of the distributed object. The method invocation to the interface object is converted to an invocation message by a *stub object*, and sent to the corresponding implementation object by a *DOVE run-time system*. On the opposite side, the receiver's run-time system unmarshals the invocation message, and invokes the appropriate method of the target implementation object through a *skeleton object*. A reply message is sent from the implementation object back to the interface object, and returned like a normal function call. This mechanism, which is called remote method invocation, allows transparent access to a distributed object irrespective of whether it is resides in local or remote space. (See figure 2.) In DOVE, a distributed object behaves either as a client or a server to interact with other distributed objects. In other words, it executes implementation object for incoming remote method invocation requests, while during the execution of the implementation object, it generates a remote invocation to other distributed object through an interface object as a client. In DOVE, *object manager* exists per host and it provides a set of indispensable services, such as object life control including creation, location and deletion of object, naming service and event propagation for fault notification, to build a transparent and easy-to-use virtual computing environment. A set of object managers constitutes a single object group which determines the domain of the virtual computing environment that might encompass a huge number of machines and networks.

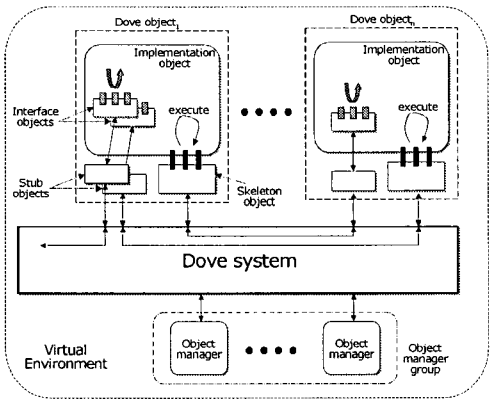


Fig. 2 DOVE object model

3.1 Object definition and instantiation

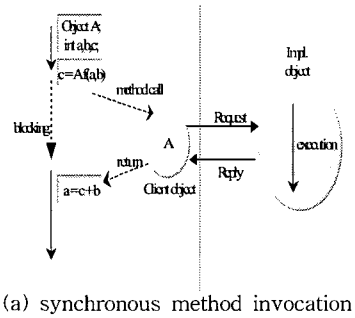
In DOVE object model, a distributed object is

defined using *Interface Description Language(IDL)*. We have adopted CORBA IDL to define distributed objects and an IDL compiler was developed to automatically generate codes for stub and skeleton objects. In DOVE object model, inheritance is supported using aggregation of interfaces with derived object. Type information is provided by *is_derived_from()* and *is_type_of()* methods for the run-time type identification of a distributed object. An interface object can be bound to any implementation object of sub-type as well as one of the same type.

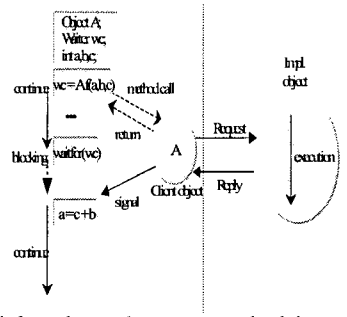
A distributed object is instantiated as a process either by a user at console or by an object manager on its local machine. When a user creates an interface object, its counterpart, an implementation object, is automatically created by the object manager, and it is connected to the interface object. The name of a distributed object can be specified at the creation of an interface object so that the newly created interface object is connected to the existing implementation object. Internally, the *object reference* which is acquired from its local object manager using name service, is used to connect an existing object. An object reference is a kind of virtual address which is used to identify and to access an implementation object. Deletion of an interface object dose not mean the deletion of its implementation object, instead the interface object is simply disconnected from the implementation object. An implementation object deletes itself after its life time expires. *Life time* of distributed object will be covered in the next section.

3.2 Remote method invocation

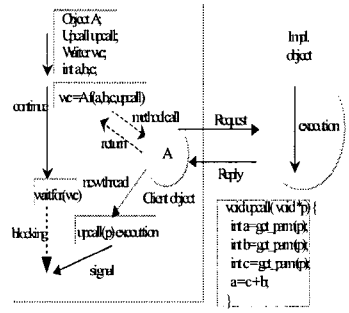
DOVE provides three kinds of remote method invocation; *synchronous*, *deferred synchronous* and *asynchronous* method invocations. In the synchronous method invocation, the sender is blocked until the corresponding reply arrives. In the deferred synchronous call, the sender can do other work immediately without awaiting the reply of the remote method invocation, but later at some point must wait for the reply in order to use the return values. In the asynchronous method invocation, the sender can proceed without awaiting the reply similarly as in deferred synchronous one, but the corresponding *upcall* method is invoked on the arrival of its reply. These communication types are not the new ones, and often used to acquire high performance in distributed system.



(a) synchronous method invocation



(b) deferred synchronous method invocation



(c) asynchronous method invocation

Fig. 3 Three types of remote method invocation

In DOVE, these types of method invocations are provided with more ease and intuitive way. A synchronous method invocation has no constraint on its return type, and is identical to the general method

invocation to local object. (See figure 3.a.) With deferred synchronous and asynchronous calls, their return type should be *void*. Actually, a *Waiter* value is returned for both method invocations, and used for synchronization between two threads one of which issues the remote method invocation, and the other returns reply respectively. In the deferred synchronous method invocation, a sender uses *Waiter* value to wait for the arrival of the corresponding reply. (See figure 3.b.) In the asynchronous mode, an Upcall method, specified when a remote method invocation is issued, is invoked using a new thread generated automatically on the arrival of its reply. In this case, *Waiter* is used for waiting for the end of *Upcall* method, not for the arrival of reply. (See figure 3.c.)

3.3 Object group

In distributed systems, group communication pattern is often used, since it provides simple and powerful abstraction. Group communication is more complicated than traditional point-to-point communication, but has recently received much attention in conventional communication mechanisms like the message passing system. In DOVE, the group communication mechanism is supported by introducing a new construct, *object group*, as a means of grouping objects and naming them as a unit for remote method invocations. An interface object can be bound to an object group, and a method invocation issued on the interface object is transparently multicast to each implementation object in the group. The interface object to an object group has the same interfaces as the one to the single object, and provides an interaction point with multiple objects in the object group so that user treats it just like a single object. Therefore, the concept of object group allows users to do more simple programming, and to have chance to get better performance if the underlying communication layer supports multicasting facilities. Even though the communication layer does not support any multicasting functions, a method invocation to the object group can be emulated by iterative invocations to each object in the object group with some loss in performance.

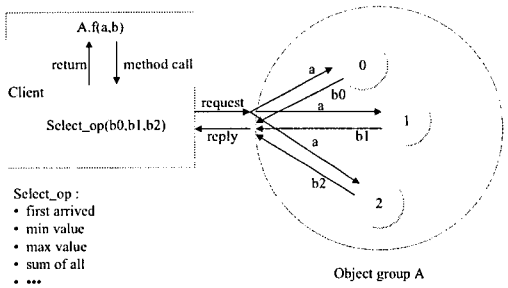
An object group can be created at any time by creating an interface object with GROUP flag. It is identified by its user-defined name and class name of the interface object. The user-defined name can be specified as a parameter when the group is created. It is implicitly deleted according to their life time policy which is specified when the group is created. Distributed objects may join or leave their groups at any time by issuing *join()* or *leave()* methods on the interface object that is bound to the group. Each object group may consist of objects with different type, but should have one common type from which they are inherited, and only the methods inherited from the common type can be invoked. In other words, an implementation object can join a group with same or super-type of it.

Object group is also used as a basic unit of inter-object synchronization. Any object can issue a *barrier()* method to the object group to which it belongs, and the *barrier()* method will be blocked until a given number of members in the object group issue the *barrier()* method.

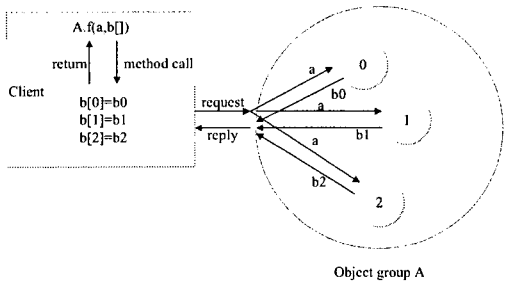
3.4 Multiple method invocation

A remote method invocation issued to an object group, or *group RMI*, is transparently multicast to each object in the object group. Since the group RMI may return one reply message per each member object in the group, a user might decide what replies to select from them.

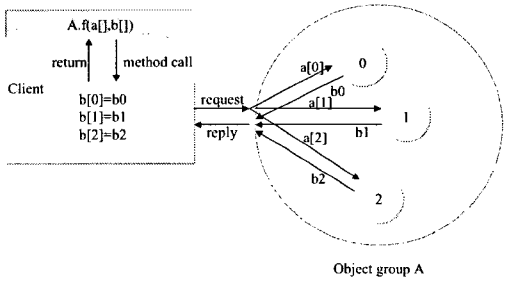
In DOVE, three types of multiple method invocations are supported for the remote method invocation to the object group: *multicast/select*, *multicast/gather* and *scatter/ gather*. A multicast/select invocation is returned with only one reply which is obtained by applying operations such as minimum, maximum, and sum to the replies of objects in the group. (See figure 4.a.) A multicast/gather invocation multicasts the same request to each member in the object group, while a scatter/gather invocation multicasts the different requests to each member by storing then in the array. (See figure 4.b and c.) Both of multicast/gather and scatter/gather method invocations are returned with arrays which store each reply from the members in the object group.



(a) multicast/select multiple method invocation



(b) multicast/gather multiple method invocation



(c) scatter/gather multiple method invocation

Fig. 4 Three types of multiple method invocation

3.5 Asynchronous event/exception handling

In DOVE, it may arises dynamically a lot of events: for example creation and deletion of new objects, change of group membership, and etc. Although most of these system events can be handled automatically by DOVE, it is frequently necessary to notify user of the occurrence of system events as well as user defined events so that user can take care of them directly.

DOVE supports asynchronous events handling by providing users with *announce()* and *notify()* methods. A *notify()* method registers or deletes event handling functions for the event with the specific type, and asks its local object manager to notify the caller of the occurrence of the event with the given type. When an event has occurred, *announce()* method sends an event to the local object manager which in turn broadcasts it to all the other object managers. Then, each object manager sends an event message to the object which has asked to notify the event using the *notify()* method.

Exception handling is similar to event handling. However, an exception differs from an event in that the former is delivered to the caller of the function where it is raised, while the latter is propagated to all the objects which desire to be notified of it. Some exceptions, for example, *NoResponse* that the target object dose not respond, are forwarded to all the other object managers to notify its fault.

4. DOVE object manager

An important design issue for a distributed or clustered system is that it provides a consistent and uniform view of how to organize applications built on top of it. And we have described our object model based on remote method invocation. Despite that our model is enough provide a uniform and single-system view, it is necessary to support additional services in order to complete distribution transparencies. Moreover, these services should be integrated with the model in such a way that all aspects related to distribution of data, computation and coordination are effectively hidden from users.

For these purpose, we introduce a special distributed object which is called *object manager*. Object manager is responsible for object life control such as creation, location and deletion of objects on its local host, naming service for binding object name to its physical address and event propagation to other object managers, which is a basic mechanism for fault notification. All the object managers constitutes a single object group which determines the domain of the virtual environment.

4.1 Object life time

An object can be created by a remote user at

console or by the object manager on its local host with three different life time: *transient*, *fixed-time* and *permanent*. A transient object is created by its local object manager when an object creation request is arrived from other object which desires to create the transient object at remote site. When an interface object is newly created, it asks its local object manager to create an implementation object of the same type with it, and the object manager cooperates with other object managers to create an implementation object on a remote site. After an implementation object is created, an object reference of the object is returned to the interface object, and is used to connect to the implementation object. Life time of a transient object expires as soon as its creator has disconnected from it, and the object deletes itself when no other objects make a reference to it. A fixed-time object is created with its life time either by its local object manager or by a user at console, and deleted after its life time has passed. A permanent object is created by a user at console, and survives until the user deletes it explicitly at a console. Life time of object group is similar to that of object. A transient group is also deleted when it is empty.

4.2 Name service

An object can have a name given by user, i. e. an alias represented by user-defined string when it is created. It is much easier and more user friendly to use an alias for object instead of its physical address. Every named object is identified by its object identifier which consists of its name and class name from which it is instantiated. Usually, a named object is created with a long fixed life time or with a permanent life time, while an unnamed object is created with a transient life time policy.

Binding from object identifier to its physical address is represented as simple triple: an object name, a class name, a physical address. The object managers keeps track of binding information about objects and object groups on its local host. Each object has its local cache to store binding information about its currently accessing objects, and first consult its local cache for binding information. If it fails, it invokes the *get_binding()* method to its local object manager. If the object manager does not contain the binding information, it multicasts the *get_binding()* method to the object manger group to obtain the binding information.

4.3 Fault tolerance

DOVE supports asynchronous fault notification using a asynchronous event handling mechanism. As a way around the impossibility of reaching consensus in an asynchronous distributed system with faulty processes[12], object managers incorporate a service similar to *Failure Suspector Service(FSS)*[13] for detecting failed object and for propagating failure beliefs to other object in consistent manner. Our service guarantees that if one object is detected as failure, the other objects will be informed of the failure. It is possible that our service makes an incorrect failure detection, meaning that under certain conditions a correct object is misjudged to have failed. However, this does not cause any problem as long as this belief is propagated to the remaining objects.

Timeout mechanism is used to detect failures, whenever a method is invoked to an object. An exception, called *NoResponse* is generated if there arrives no response within timeout period, and sent to its local manager which in turn forwards it to all the other object managers along the global paths for event propagation as described in the previous section. Similarly, the fault of an object manager can be detected by another object manager.

5. DOVE system

In this section, we deal with the design of DOVE run-time system, called DOVE system, compliant with DOVE object model described in the previous section. We propose a portable and flexible system architecture which offers invariable system interfaces independent of underlying platform, and which consists of three layers that interact with each other using the system interfaces.

5.1 Multi-layered architecture

DOVE system consists of three layers: method invocation layer, message passing layer and communication layer. (See figure 5.) It provides a set

of interfaces which are platform independent by decoupling method invocation layer from communication layer.

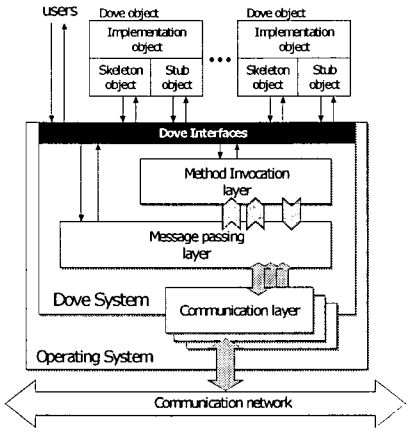


Fig. 5 Multi-layered architecture of DOVE system

Normally, a user program interacts with DOVE through stub objects and skeleton objects. Stub and skeleton objects are generated by a DOVE IDL compiler automatically to provide user with interfaces through which methods can be invoked. User can make use of different interfaces for the same operation to support diverse method invocations and group method invocations. The calling semantics of method invocations are determined by their parameters. A method invoked through a stub object is converted into an invocation structure which is then passed to the method invocation layer.

The method invocation layer marshals each invocation structure into an invocation message, and then passes it to message passing layer after registering it into an invocation table in order to deal with the reply message properly. When it receives an invocation message from remote site, it creates a new thread for executing the method of the object which corresponds to the invocation message. When a reply message is returned to method passing layer, it is unmarshaled into an invocation structure, and then matched with the previously registered one by scanning through the invocation table to perform the proper action according to the semantic of the method invocation.

The message passing layer carries out object name binding to physical address, and delivers messages to distributed object using one of the underlying several communication layers. Each communication layer can be implemented using any distributed protocol which makes use of different interfaces and naming schemes. The main purpose of the message passing layer is to encapsulate the method invocation layer from communication layers. The message passing layer behaves like an adaptor that can be plugged into a specific communication layer, and decouples the method invocation layer from the communication layer in order to provide the method invocation layer with uniform interfaces for message delivery and object group membership. The message passing layer stores information about group membership, and takes care of remote method invocations to the object group by using multicasting function if the communication layer supports it; otherwise by iterative execution of an unicast function in the communication layer.

In order to achieve the full functionality of DOVE system, the communication layer should be equipped with reliable unicast, reliable multicast and process group management. The minimal requirement for the communication layer is reliable unicast like TCP/IP. Currently, DOVE has installed two communication layers, one for reliable unicast using TCP/IP and another for reliable and totally ordered group communication using IP multicast.

Each layer of DOVE system has its own thread of control and works concurrently. The multi-layered architecture of DOVE provides users with more extensibility, since each layer is implemented as an independent module which interacts with other layers through uniform interfaces.

5.2 Data marshal and unmarshal

In a remote method invocation, its parameter and a return value must be marshaled before they are transmitted over the network to provide heterogeneity in a networked environment. There is no dispute about marshaling simple data types. Rather, we are

concerned about how a user-defined, complex data structure in the remote method invocation can be transmitted. Some systems, for instance, Modula-3, use run-time type information(RTTI) to marshal user-defined data types, while Legion and PVM enforce the programmer to write codes to marshal and unmarshal user-defined data types.

In DOVE, every data type, built-in or user-defined data types, is inherited from single *BaseType* class with *marshal()* and *unmarshal()* methods. DOVE IDL compiler automatically generates codes for marshal and unmarshal methods for built-in data type by using CDR(Common Data Representation)[1] specification, and for user-defined data type by calling marshal and unmarshal methods recursively for each of its attributes until a built-in type is met.

5.3 IDL compiler

In DOVE system, the interfaces of distributed objects are defined by CORBA compliant interface description language(IDL). DOVE IDL compiler is designed so that by compiling IDL, a pair of stub and skeleton objects is automatically generated for each *interface* of IDL in C++ code using DOVE class library.

There are two types of interfaces for a remote method invocation according to its return type: non-*void* return type and *void* return type. The former is mapped into a synchronous remote method invocation, and the other into either a deferred synchronous or an asynchronous method invocations at run time according to whether *UpCall* parameter is specified or not respectively. That is, an interface with *UpCall* parameter is treated as asynchronous method invocation, and the one without *UpCall* parameter as deferred method invocation.

Also, there are three kinds of interfaces according to its parameter type. In a simple remote method invocation and a multiple method invocation of multicast/select type, input and output parameters are represented in single variable respectively, while in a multiple method invocation of multicast/gather represented in single variable and sequence, or array of variable size, and in a multiple method invocation of scatter/gather both in sequence, as in figure 6.

```
// CORBA IDL
interface Renderer {
  void render( in Ray ray, out Color color);
 };

 // C++ mapping
 class Renderer {
 public:
  // unicast and multicast/select call
  Waiter render(Ray ray, Color& color, UpCall = NULL);
  // multicast/gather call
  Waiter render(Ray ray, ColorSeq color, UpCall = NULL);
  // scatter/gather call
  Waiter render(RaySeq ray, ColorSeq color, UpCall = NULL);
      }
```

Fig. 6 Example of IDL to C++ mapping

6. DOVE implementation

6.1 DOVE class library

DOVE is a multi-layered architecture which consists of several layers: a method invocation layer, a message passing layer and communication layers. It is implemented in C++ class library with a set of interfaces which are independent on the underlying platform. *MIL* class is designed for the method invocation layer, *MPL* class for the message passing layer, and *Comm* class for the communication layer. The *MIL* class provides interfaces for invoking methods, returning replies, creating a thread for the execution of the implementation and handling exception and event. The *MPL* class has a set of interfaces for asynchronous message passing between *MIL* and *Comm* classes. The *Comm* class encapsulates the underlying protocols from message passing layer, and provides uniform interfaces for sending and receiving data.

In addition to the classes for each layer, *DoveObject* and *DoveServant* is designed for stub and skeleton object respectively. *DoveObject* is a base class for stub objects. Every stub object for each type of distributed object must be derived from the *DoveObject* class, which provides basic functions such as connecting to and disconnecting from a distributed object, generating an invocation structure

and invoking a remote method to a distributed object by passing the invocation structure to the method invocation layer. *DoveServant* is a base class for skeleton object. Every class for skeleton object should inherit the *DoveServant* class, and redefine *dispatch()* method for proper execution of user-defined methods in the distributed object. The *DoveServant* class exports methods for activation and deactivation of distributed object. Activated objects are registered into servant table in the method invocation layer. When a request message is received, it is matched with the previously registered one, and the *dispatch()* method is executed. Note that a normally user need not write code for stub and skeleton objects, but the DOVE IDL compiler generates them automatically. Since a user and each layer of DOVE interacts with one another using the interfaces in the class library without directly accessing the underlying system, it can be built on various heterogeneous machine without any change in its implementation. Therefore, DOVE provides a portable and flexible virtual environment which can be easily extended, and adapted to a new technology.

6.2 DOVE system interfaces

A set of interfaces are supported by a *Dove* class as static member functions. The *Dove* class encapsulates the layered architecture and provides users and layers with a number of system interfaces including system initialization and shutdown, thread creation and synchronization, message passing, object group management, object activation and deactivation, event notification and announcement, method invocation, and data marshalling and unmarshalling. Usually, users dose not confront with these interfaces directly, instead they interact with DOVE system through stub and skeleton objects encapsulating DOVE system interfaces. A complete list of system interfaces is shown in figure 7. In the figure, Handle represents local connection point to other object or object group, and IOR stores object reference of an object or a group. DoveMessage, DoveBuffer and DoveEvent are used for sending messages, storing marshalled data and specifying events respectively.

6.3 DOVE monitor

```
class Dove {
 // machine init & shundown
 int init(int argc, char* argv[]);
 void shutdown();

 // thread create and synchronization
 int create_thread(const Thread&, void*);
 int thread_exit();
 int signal(Waiter&);
 int waitfor(Waiter&);
 int waitall(Waiter[], int);
 int waitany(Waiter[], int);

 // servant activation
 void activate(DoveServant* impl);
 DoveServant* deactivate();

 // method invocation
 void invoke(DoveInvocation* inv);

 // connection handling
 Bool connect(const IOR* dest, Handle& h);
 void close(const Handle& h);

 // message passing
 int send_message(DoveMessage*, const Handle& to);
 int recv_message(DoveMessage*, Handle& from);

 // event handling
 int announce(const DoveEvent& ev);
 int notify(char* etype, const EventHandler& evh);

 // group management
 int create_group(Handle&,char* cname, char* gname);
 int delete_group(const Handle& grp);
 int join_group(const Handle& grp);
 int leave_group(const Handle& grp);

 // data marshal & unmarshal
 void marshal(DoveBuffer*, const BaseType*);
 void unmarshal(DoveBuffer*, BaseType*);
 };
```

Fig. 7 DOVE system interfaces

A monitor program is developed to display the current states of DOVE. This tool allows a user to view which machines are currently joined into DOVE, which objects are installed and running on the hosts. User can create and execute a new object from the installation list, and delete an object from the list of running objects at console. The monitor program is an application running in DOVE, and able to interact

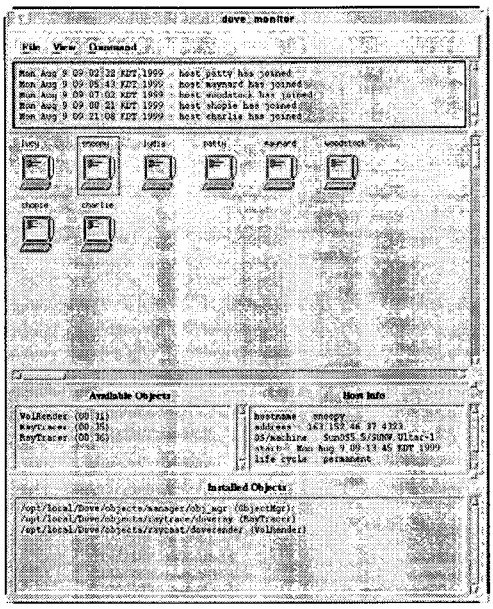


Fig. 8 DOVE monitor user interface

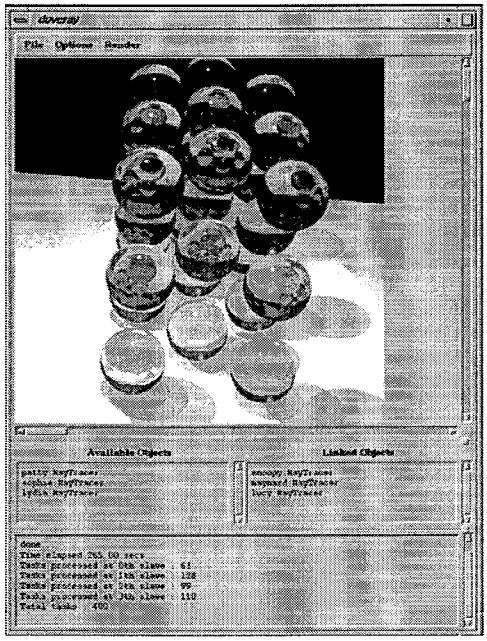


Fig. 9 ray-tracing task manager

with its local object manager to allow users to control DOVE interactively. The user interfaces of the monitor is shown in figure 8.

7. Evaluation

For the performance evaluation purpose of DOVE, a parallel ray-tracing software is developed on both DOVE and PVM which consists of 17 heterogeneous machines, two Ultrasparc1, two SGI O2, a SGI Octane, 9 Pentium II PCs running Linux and three Pentium II PCs running Windows NT/98 connected by 100Mbps Ethernet. Ray tracing is a simple and powerful technique for rendering realistic images by tracing all the rays from eye through each pixel in the view plane. Since it requires high computation time for the calculation of intersections between rays and objects as well as the tracing of all the reflected and refracted rays, it has been considered as an adequate application for parallel computation. A parallel ray-tracing software is designed in master/slave scheme where one master task manager object interacts with ray tracer objects. Master/slave paradigm suits well virtual computing environment since its dynamic nature of job scheduling which can reduce the unbalance of computational power among the heterogeneous machines. One master task manager object is running on a Ultrasparc1 work-station for job distribution, and each of ray tracer objects on the different machine. Task manager object schedules the assignment of pixels to ray tracer objects which in turn calculate the value of each pixel. Figure 9 shows a user interface for task manager object with the display of its rendering result. An IDL description for ray tracer object and a simple code for task scheduling at task manager object are shown in Figure 10 and 11 respectively. A

```
interface RayTracer {
   struct Task {
        long sx, sy;          // start point of task
        long width, height;   // width and height of task
        sequence<octet> r, g, b;   // rendering results
    };
    void render( inout Task task);
    void setup( in string filename,
                in long screen_w, in long screen_h);
};
```

Fig. 10 IDL description for RayTracer object

Table 1 Measurement of relative performance

machines	M1	M2	M3	M4	M5	M6	M7
spec.	Linux (PII-300)	Win98 (PII-300)	Win98 (PII-366)	WinNT (PII-300)	Sol 2.5 (USparc1)	IRIX 6.3 (O2)	IRIX 6.5 (Octane)
running time(sec)	427.475	312.460	253.670	307.602	858.340	781.636	466.224
relative perf.	1.0	1.368	1.685	1.390	0.498	0.547	0.917

Table 2 Speedup with respect to number of ray tracers

number of ray tracers		1 (M1)	2 (M1)	4 (M3,5)	8 (M1,1,4,6)	12 (M1,2,6,7)	16 (M1,1,1,1)
expected speed up		1.0	2.0	4.183	8.120	11.952	15.952
DOVE	running time (sec)	427.475	217.300	104.298	54.935	37.790	29.630
	speed up	1.0	1.967	4.099	7.781	11.312	14.931
	efficiency (%)	100.0	98.36	97.98	95.83	94.64	93.60
PVM	running time (sec)	427.475	227.622	107.803	56.217	38.566	29.336
	speed up	1.0	1.878	3.965	7.604	11.084	14.572
	efficiency (%)	100.0	93.90	94.80	93.65	92.74	91.35

* Parenthesis in number of ray tracers represents a machine used for running an additional ray tracer object.

```
// task scheduling example
Task task[4]; Waiter w[4];
// create 4 raytracer objects
RayTracer* rt = new RayTracer[4];
for(int i=0;i<4;i++) {
    rt[i].setup("crystal",500,500);
    make_task(task[i]);
    w[i] = rt[i].render(task[i]);
}
while ( more_task_left ) {
    n = waitany(w,4);     // returns an index of w
[]
    draw_task(task[n]);
    make_task(task[n]);
    w[n] = rt[n].render(task[n]);
    ...
}
```

Fig. 11 Main code for ray tracing using RayTracer objects

user can obtain a stub and a skeleton object by compiling this IDL description using DOVE IDL compiler.

Since each machine has different computing powers, we have measured relative performance with respect to Linux machine. Then, the expected(or ideal) speed up can be computed as a sum of each relative performance value of participating machines. The relative computing powers are shown in table 1. Table 2 shows the execution time of ray-tracing according to the number of ray tracers. Efficiency is calculated as the ratio between the actual and ideal. As the number of ray tracer increases, DOVE shows an almost linear speedup better than PVM. This is due to the fact that in DOVE, computation and communication is efficiently overlapped through asynchronous remote method invocation, and that DOVE executes a remote method invocation directly

to the object without passing through any intermediate object manager, while in PVM, all the message from different processes are routed through daemon process which runs in each host, producing overhead in daemon process.

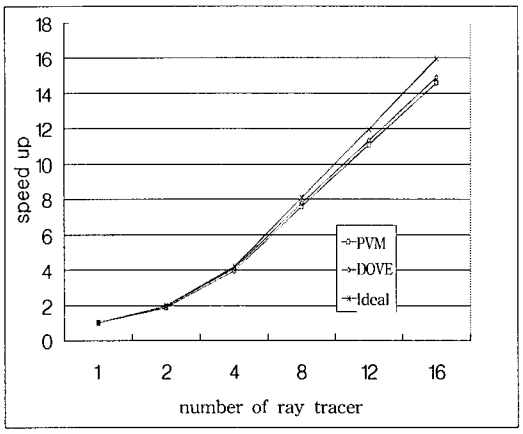


Fig. 12 Speed up vs. number of raytracer objects

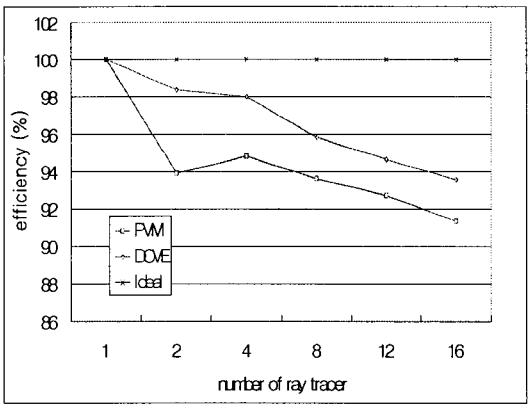


Fig. 13 Efficiency vs. number of raytracer objects

8. Conclusion

In this paper, we have presented a new object-oriented distributed computing environment called DOVE which is based on a distributed object model and designed to build an easy-to-use virtual programming environment for parallel applications as well as distributed ones. Efficient parallelism is supported by diverse remote method invocation, multiple method invocation for object group, multi-threaded architecture in the server implementation and diverse synchronization schemes. Heterogeneity is achieved by automatic data marshalling and unmarshalling, and an easy-to-use and transparent programming environment is provided by stub and skeleton objects generated CORBA compliant IDL compiler, object and group management and naming service by object manager. Autonomy of distributed objects, multi-layered architecture and decentralized approaches in hierarchical naming service and object management makes DOVE more extensible and scalable. Also, fault tolerance is provided by object fault detection using a timeout mechanism and fault notification using asynchronous exception handling methods.

Currently, DOVE is implemented as a C++ class library, and a user can develop a parallel or distributed applications on a heterogeneous environment using the class libraries as if it resides in one virtual computer. For the experiment of DOVE, a parallel software for ray tracing has been developed for both of DOVE and PVM which consists of Unix workstations, Linux and Windows NT/98 machines, and its performance was compared with that of PVM. Our experience shows that DOVE can provide an easy-to-use virtual programming environment by supporting efficient parallelism, heterogeneity, transparency, stub and skeleton objects generated by IDL compiler, object management and name service by object manager, and that its performance is better than that of PVM.

With DOVE, user can easily form a parallel virtual programming environment at low cost by sharing the existing resources and connecting them in a network. Since the computer networks are becoming larger and faster, and the performance of machines more powerful these days, faster and more powerful, high performance machines over a local or wide-area networks can be connected to make a virtual parallel computer with the performance nearly equal to supercomputer at low cost. Therefore, we believe that DOVE, a pioneering work for virtual computing for the first time in domestic, will make a significant research contribution to the parallel applications for

computation intensive problems as well as many other distributed applications which requires a virtual environment like collaborative work, distributed interactive simulation, virtual reality, and etc.

As a future work, we are going to build a large-scale DOVE test-bed with comprises a variety of institutions over a wide area network, and to develop resource management schemes which can efficiently support the large scale test-bed. We hope that DOVE can be used as a research infrastructure for users developing parallel or distributed application software or cooperating with each other at remote sites.

Reference

[1] Object Management Group, Inc. (OMG), *The Common Object Request Broker: Architecture and Specification*, OMG Document Revision 2.2, February 1998.

[2] T. B. Downing, *Java RMI : Remote Method Invocation*, IDG Books worldwide, 1998

[3] E. Frank and III Redmond, *DCOM : Microsoft Distributed Component Object Model*, IDG Books worldwide, 1997

[4] R. Manchek, *Design and Implementation of PVM version 3*, Master's Thesis University of Tennessee, June 1994.

[5] A. Geist, A. Beguelin and et. al., *PVM 3 User's guide and Reference manual*, ORNL/TM-12187, September 1994.

[6] MPI Forum, "MPI: A message-passing interface standard," International Journal of Supercomputer Application, 8(3/4):165-416, 1994.

[7] M. Lewis and A. Grimshaw, "The Core Legion Object Model," University of Virginia Computer Science Technical Report CS-95-35, August 1995

[8] K. P. Birman, "The Process Group Approach to Reliable Distributed Computing," Communication of ACM, 36(12), December 1993.

[9] K. P. Birman and R. Van Renesse, "Reliable Distributed Computing with the Isis Toolkit," IEEE Computer Society Press, 1994

[10] Y. Amir, D. Dolev, S. kramer and D. Malki, "Transis: A Communication Sub-system for High Availability," 22nd International Symposium on Fault-Tolerant Computing, IEEE, July 1992.

[11] G. Booch, *Object Oriented Analysis and Design with Applications*, The Benjamin/Cummings Publishing Company, Inc., 1994.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Concensus with One Faulty Process," Journal of ACM, 32(2), April 1985.

[13] A. Ricciardi, A. Schiper, and K. P. Birman, "Understanding Partition and the 'No Partition' Assumption," Technical Report 93-1355, Department of Computer Science, Cornell University, June 1993.

김 형 도

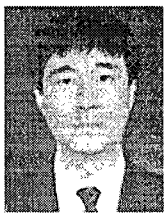
1993년 2월 고려대학교 전자공학과 공학사. 1995년 2월 고려대학교 전자공학과 공학석사. 2000년 2월 고려대학교 전자공학과 공학박사. 관심 분야는 분산 및 병렬처리 시스템

우 영 제

1992년 2월 고려대학교 전자공학과 공학사. 1997년 2월 고려대학교 전자공학과 공학석사. 1997년 2월 ~ 현재 고려대학교 전자공학과 박사과정. 관심 분야는 분산 시스템, mobile object

류 소 현

1998년 2월 고려대학교 전자공학과 공학사. 2000년 2월 고려대학교 전자공학과 공학석사. 2000년 2월 ~ 현재 고려대학교 전자공학과 박사과정. 관심 분야는 분산 처리 및 프로토콜



정 창 성

1981년 서울대학교 전기공학과 졸업. 1984년 Northwestern University Dept. of Computer Science 석사. 1987년 Northwestern University Dept. of Computer Science 박사. 1987년 ~ 1992년 포항공대 전산학과 조교수. 1992년 ~ 1997년 고려대학교 전자공학과 부교수. 1998년 ~ 현재 고려대학교 전자공학과 정교수. 관심 분야는 병렬/분산 알고리즘, 병렬구조, 병렬/분산 simulation, visual information processing