

# Class Management in a Distributed Actor System

by

Venkateswara Rao Vykunta

Project Report submitted to the faculty of the  
Virginia Polytechnic Institute and State University  
in partial fulfillment of the requirements for the degree of

**MASTER OF SCIENCE**

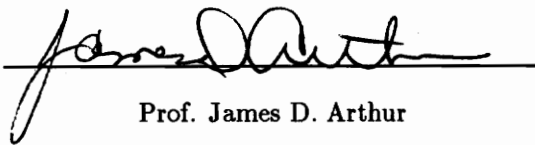
in

Computer Science

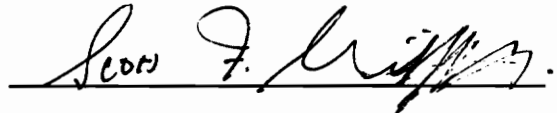
APPROVED:



Prof. Dennis Kafura, Chairman



Prof. James D. Arthur



Prof. Scott F. Midkiff

July, 1994

Blacksburg, Virginia

C. 2

LD  
5655  
V851  
1994  
V958  
C.2

# **Class Management in a Distributed Actor System**

by

Venkateswara Rao Vykunta

Committee Chairman: Prof. Dennis Kafura

Department of Computer Science

## **(ABSTRACT)**

The goal of this project is to develop part of an environment to allow the creation of distributed applications in ACT++. ACT++ is a distributed, object-based programming system in which concurrent object-oriented programs can be written in C++. Programs in ACT++ consist of a collection of active objects called actors. This project is concerned only with the problems related to the creation, destruction, and the invocations of the methods of an object located through a directory agent on a remote machine. The specific services developed in the project allow the dynamic loading and unloading of object code for a class when it is not available in memory, static loading of object code on machines that do not support dynamic linking, and the location of classes and their instances through a directory agent. Other related projects concern the development of a communication infrastructure that configures a collection of heterogeneous machines for use in the distributed version of ACT++ and the use of lightweight threads. The solution functionally was tested on several simple examples.

# TABLE OF CONTENTS

<b>1</b>	<b>INTRODUCTION</b>	<b>1</b>
<b>2</b>	<b>DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS</b>	<b>2</b>
2.1	Objects and Object-Based Programming Languages . . . . .	2
2.2	Why Object-Oriented Systems ? . . . . .	4
2.3	Object-Based Programming Systems . . . . .	5
2.4	Distributed Object-Based Programming Systems . . . . .	6
2.5	Object Structure . . . . .	7
2.5.1	Granularity . . . . .	8
2.5.2	Composition . . . . .	8
2.6	Object Interaction Management . . . . .	10
2.6.1	Locating an Object . . . . .	10
2.6.2	System-Level Invocation Handling . . . . .	12
2.6.3	Detecting Invocation Failures . . . . .	13
2.7	The CORBA Architecture . . . . .	14
2.7.1	The Structure of ORB Interfaces . . . . .	14
2.7.2	Interoperability . . . . .	16
<b>3</b>	<b>THE ACTOR MODEL</b>	<b>18</b>
3.1	Description of the Actor Model . . . . .	18
3.2	ACT++: An Implementation of the Actor Model . . . . .	20
3.3	A Distributed Actor System . . . . .	21
<b>4</b>	<b>AN EXECUTION ENVIRONMENT FOR ACT++</b>	<b>24</b>

*CONTENTS*

4.1	Overview of the Environment . . . . .	24
4.2	Details of the Current Implementation . . . . .	26
4.2.1	The Communications Interface with the Server . . . . .	26
4.2.2	The List Manager Class . . . . .	31
4.2.3	The Handle Class . . . . .	33
4.2.4	The Map Class . . . . .	34
4.2.5	The Directory Services . . . . .	34
4.2.6	Dynamic Loading Mechanism in Distributed ACT++ . . . . .	39
4.2.7	Static Loading Mechanism in Distributed ACT++ . . . . .	42
4.2.8	Functional Testing . . . . .	43
<b>5</b>	<b>CONCLUSIONS AND FUTURE WORK</b>	<b>48</b>

## LIST OF FIGURES

2.1	Performing an action in the passive object model. . . . .	9
2.2	Performing an action in the active object model. . . . .	10
2.3	A request being sent through the Object Request Broker. . . . .	15
2.4	The structure of Object Request Broker interfaces. . . . .	15
3.1	An abstract representation of an actor. . . . .	18
3.2	An abstract representation of transition. . . . .	19
3.3	A centralized actor system. . . . .	21
3.4	A distributed actor system. . . . .	23
4.1	An overview of the environment. . . . .	25
4.2	The actor constructor message format. . . . .	26
4.3	The actor construction process. . . . .	27
4.4	The actor method invocation message format. . . . .	29
4.5	The actor method invocation process. . . . .	29
4.6	The reply message format. . . . .	30
4.7	The reply process using cbox directory. . . . .	30
4.8	The actor destructor message format. . . . .	31
4.9	The actor destructor process. . . . .	32
4.10	The testing environment. . . . .	44
4.11	Testing the components of the environment . . . . .	45

## **ACKNOWLEDGEMENTS**

I wish to express my most sincere gratitude and appreciation to Dr. Dennis Kafura for his guidance and patience throughout the development of this project. I thank Dr. James Arthur and Dr. Scott Midkiff for serving on my committee and giving suggestions to improve my project report.

I dedicate this project to my parents for their support, mentorship and unflagging faith in myself.

# Chapter 1

## INTRODUCTION

With the widespread use of computers, software applications are becoming more complex and are increasing in size. Computing environments are becoming increasingly distributed and heterogeneous and hence the development of an environment supporting both machine and language heterogeneity would decrease the cost of development of distributed applications, and also increase the interoperability of systems. The use of the object-oriented paradigm can help in creating such an environment. ACT++ is one such distributed, object-based programming system in which concurrent, object-oriented programs can be written in C++.

This project is an attempt to create part of an environment to support the distribution of an actor system in an heterogeneous environment. The specific services developed in this project allow the dynamic loading and unloading of object code for a class when it is not available in memory, static loading of object code on machines that do not support dynamic loading and the location of classes and their instances in memory through a directory agent.

In the remainder of the report, Chapter 2 presents background on distributed, object-based programming systems. Chapter 3 presents the actor model. An implementation of this model, ACT++, is also described. Chapter 4 presents an overview of the environment created, the different components involved and their implementation. Finally, Chapter 5 discusses the limitations of the environment created. Some enhancements are also suggested for future work.



## Chapter 2

# DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS

The development of distributed operating systems and object-based programming languages makes possible an environment in which objects may execute concurrently on a collection of loosely coupled processors. An object-based programming language supports the design and creation of programs as a set of autonomous components, whereas a distributed operating system permits a collection of machines to be treated as a single entity. The combination of these two concepts has resulted in systems that are referred to as *distributed, object-based programming systems*.

This chapter discusses the issues in the design and implementation of such systems. Following the presentation of fundamental concepts and various object models, issues in object structure, object interaction management and the CORBA architecture are discussed.

### 2.1 Objects and Object-Based Programming Languages

Peter Wegner presents in [WEG87] different features that allow one to categorize object-based languages depending on the set of features that they provide. The first feature described is *objects*.

**Object:** An *Object* is an entity that encapsulates private *state information* or data, and a set of associated *operations* or procedures that manipulate the data.

A programming language is defined as being *object-based* if it supports objects as a language feature.

The second feature described is that of class which can be defined as follows:

## CHAPTER 2. DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS

**Class:** A class is the direct extension of the notion of an abstract data type; it is a template from which objects may be created by “new” operations. A group of objects that have the same set of operations and the same state representations are considered to be of the same class. A class maintains no state and performs no operations. Classes typically exist at compile time; objects exist at execution time.

A language that offers the objects and classes features is called a *class-based language*.

The third property provides a mechanism that permits new classes to be developed from existing classes simply by specifying how the new classes differ from the originals. This mechanism, called inheritance, is defined as follows.

**Inheritance:** A class may inherit the operations and behavior of a *base class* or *super class*, and it may have its operations and behavior inherited by a *derived class* or *sub class*. More than one class may be derived from a single base class. A super class provides functionality common to all of its subclasses, whereas a subclass provides additional functionality to specialize its behavior.

A language containing those three features is then called *object-oriented*.

Another variation of the inheritance scheme is *delegation*. Delegation is defined by Wegner as follows.

**Delegation:** Delegation is a mechanism that permits an instance object to delegate responsibility for servicing an invocation request to another object. This differs from the inheritance scheme in that it is class independent. Individual instance objects of the same class may have different objects servicing requests they are unable to service.

According to these definitions, the languages C++ [STR91], Ada 9X [Ada9X], Modula 3 [NEL], and Smalltalk [Gold] are object-oriented languages, whereas the languages Ada [Barnes], and Modula-2 [Wirth] are object-based languages.

## CHAPTER 2. DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS

### 2.2 Why Object-Oriented Systems ?

Object-based and object-oriented languages have become widely used because they directly support a design approach within which the problem is modeled as a set of multiple, interacting, autonomous entities and the relationships among them. This philosophy of dividing a program into multiple subcomponents is not new. In fact, many of these ideas come from traditional software engineering principles that stress such a design methodology. According to these principles, a program should have the following five characteristics.

- *Abstract.* The design concepts should be separated from the implementation details. A program should hide the design decisions and the data structures used.
- *Structured.* A large program should be decomposed into components of a manageable size, with well-defined relationships established between the components.
- *Modular.* The internal design of each component should be localized so that it does not depend on the internal design of any other component.
- *Concise.* The code should be clear and understandable.
- *Verifiable.* The program should be easy to test and debug. It is easy to test and debug a class separately from the other classes. As classes provide high cohesion and low coupling, almost no testing is required after the integration phase.

The object-oriented paradigm offers another great advantage in terms of *maintainability* and *reusability*. This is primarily due to two reasons:

- data abstraction, and
- inheritance

One part of maintenance deals with adding functionality to the application software. The inheritance mechanism supports this aspect of maintenance in two ways.

## CHAPTER 2. DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS

First, the inheritance relation facilitates the reuse of existing definitions to ease the development of new definitions. It reduces the code bulk by reducing the need to redevelop common functionalities and, as a consequence, increases the consistency. Second, objects become polymorphic with the use of inheritance. Polymorphism is the ability to take more than one form. In object-oriented programming, it refers to the ability of an object to refer at run-time to instances of various classes. Another feature called *dynamic binding* is needed to effectively use the polymorphism property of objects. Dynamic binding means that the code associated with a given procedure call is not known until the moment of the call at run-time. This is powerful, as it allows procedures or functions to operate on more than one type.

Concerning reusability, the object-oriented paradigm combines design techniques and language features to provide strong support for the reuse of software modules. Every time an instance of a class is created, reuse occurs. The inheritance mechanism facilitates the reuse of existing definitions to ease the development of new definitions. The development of meaningful abstractions also encourages reuse.

### 2.3 Object-Based Programming Systems

An object-based programming system is defined as a computing environment that supplies both an object-based programming language and a system that supports object abstraction at the operating system level. This enables objects to be maintained, managed, and used efficiently. It also permits objects to be shared by multiple users. In contrast, object-based programming languages do not allow objects to be shared. To provide the later functionality, the system typically assigns a unique identifier to each object so each can be uniquely specified.

The operating system of an object-based programming system supplies a global, machine-wide object space by providing facilities for the following:

- object management,

## CHAPTER 2. DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS

- object interaction management, and
- resource management

Objects are the fundamental resources of an object-based programming system. Object management facilities provide features for making the effects of an action on persistent objects permanent, for synchronizing the execution of multiple concurrent invocations with in an object, for protecting objects from unauthorized clients, and for recovering objects that fail. Object interaction management provides facilities for locating server objects, for handling object interactions, and for detecting invocation failures. Resource management provides mechanisms to manage the physical resources of the system, including primary memory, secondary storage devices, processors, and workstations of the network. This project specifically focusses on the object interaction management in the distributed actor system.

### 2.4 Distributed Object-Based Programming Systems

A distributed, object-based programming system provides the features of an object-based programming system as well as a decentralized or distributed computing environment.

According to Chin and Chanson [CC91], distributed, object-based programming systems typically have the following characteristics.

**Distribution.** A distributed, object-based programming system combines a network of independent, possibly heterogeneous, machines.

**Transparency.** The system may hide the distributed environment or other underlying details from the users. For example, it can provide the feature of *location transparency* so a user does not have to be aware of the machine boundaries and the physical locations of an object in order to use the object. The user does not have to worry whether the definition of an object is in memory or in secondary storage.

## CHAPTER 2. DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS

**Data Integrity.** The system may provide *persistent objects*, which means objects may outlast the life of an application. In this case, the system must ensure that an object is always in a state that is the result of the successful termination of an operation. If an operation does not successfully complete, the system ensures that all changes made to the object's state are undone.

**Fault Tolerance.** The failure of a workstation should not induce the failure of the whole system. The remainder of the system should be able to offer the services that are not dependent on the workstation that has failed.

**Availability.** A distributed, object-based programming system may take steps to ensure that all objects remain available to a high probability, despite workstation failures.

**Recoverability.** If a workstation fails, the system should restore the persistent objects that reside on it.

**Object Autonomy.** The system may permit the owner of an object to specify the clients that have the permission to make invocations on the object.

**Program Concurrency.** The system should be able to assign the objects of a program to multiple processors so they may execute concurrently.

**Object Concurrency.** An object should be able to serve multiple, nonmodifying invocation requests concurrently. Note that this is not true concurrency unless an object resides in a multiprocessor, since only one request can be processed at any one time.

**Improved Performance.** A well-designed program should execute more quickly than in a conventional system.

### 2.5 Object Structure

The structure of the objects supported by a distributed, object-based programming system influences its overall design. This section defines three types of objects that can

## CHAPTER 2. DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS

be supported by distributed, object-based programming systems and two ways they can be composed.

### 2.5.1 Granularity

The relative size, overhead, and amount of processing performed by an object characterizes its *granularity*.

- **Large-Grain Objects:** These objects are characterized by their large size, relatively large number of instructions they execute to perform an invocation, and relatively few interactions they have with other objects. Large-grain objects typically reside in their own address spaces. Some examples of a large-grain object are a major component of a program, a file, and a single-user database.
- **Medium-Grain Objects:** Medium-grain objects can be created and maintained relatively inexpensively; they are smaller in size and in scope than large-grain objects. Typical examples of medium-grain objects are data structures and their operations such as a linked list, a queue, and the components of a multiuser database. A number of medium-grain objects may reside in the address space of a single large-grain object.
- **Fine-Grain Objects:** Fine-grain objects are characterized by their small size, small number of instructions they execute, and relatively large number of interactions they have with other objects. Some examples of fine-grain objects are data types and their operations that are provided by conventional programming languages such as booleans, integers, and complex numbers.

### 2.5.2 Composition

The relationship between the processes and the objects of a distributed, object-based programming system characterizes the *composition* of objects. Two possible approaches are described below.

## CHAPTER 2. DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS

**Passive Object Model:** In this model, processes and objects are completely separate entities. A process is not bound nor is it restricted to a single object. Instead, a single process is used to perform all the operations required to satisfy an action. Consequently, a process may execute within several objects during its lifetime. When a process makes an invocation on another object, its execution in the object in which it currently resides is temporarily suspended. The process is then mapped into the address space of the second object, where it executes the appropriate operation. After completion, it is returned to the first object, where it resumes the execution of the original operation. The sequence of events is illustrated in Figure 2.1.

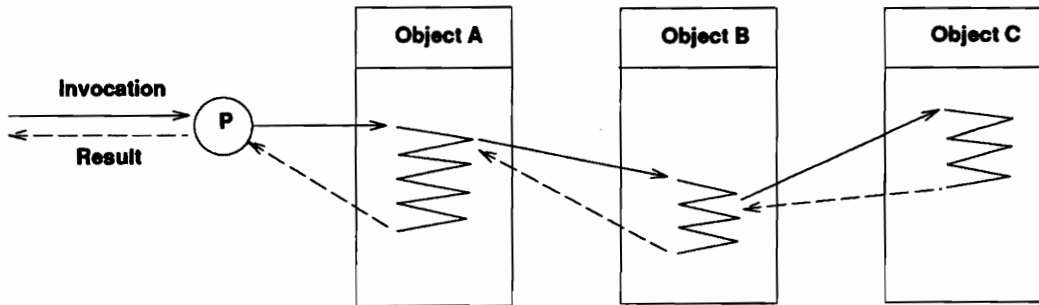


Figure 2.1: Performing an action in the passive object model.

The main advantage of using the passive object model is that there is virtually no restriction on the number of processes that can be bound to an object, thus allowing a high-level of object concurrency. The main drawback of this approach is that mapping a process into and out of the address space of multiple objects can be difficult and expensive.

**Active Object Model:** In this model several server processes are created for and assigned to each object to handle its invocation requests. Each process is bound to the particular object for which it is created. When the object is destroyed, its processes are also destroyed. In the active object model, an operation is not typically accessed directly by the calling process, as in the case of a traditional procedure call. Instead, when a client makes an



## CHAPTER 2. DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS

operation invocation, a process in the corresponding server object accepts the request and performs the operation on the client's behalf. If in the course of executing an operation an invocation on another object is made, the process issues the invocation request and waits for a result. A server process in the second object is then called upon to execute the new operation, and so on. When the operation completes, the server returns the result to the client. The sequence of events is illustrated in Figure 2.2.

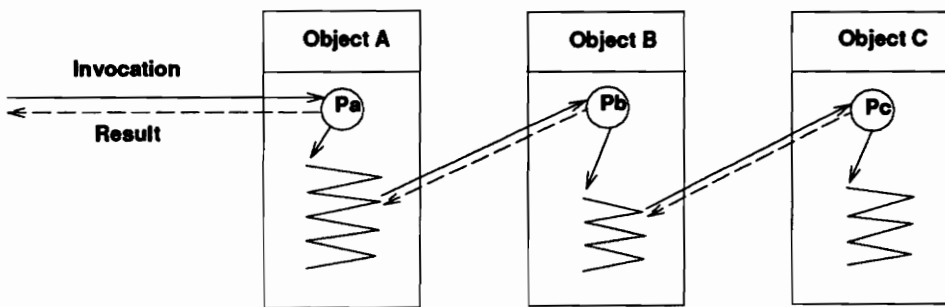


Figure 2.2: Performing an action in the active object model.

### 2.6 Object Interaction Management

A distributed, object-based programming system is responsible for managing the invocations between cooperating objects. When a client wants to invoke the method of an object, the system must locate the specified object, take the appropriate steps to invoke the specified operation, then possibly return a result back to the client.

#### 2.6.1 Locating an Object

A distributed, object-based programming system should provide the property of *location transparency* so a client does not have to be aware of the physical location of an object in order to invoke the methods of the object. Whenever an invocation is made, the system must determine which object is to be invoked and on which machine the object currently

## CHAPTER 2. DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS

resides in order to deliver the request to it. The system must assign a unique identifier to each object. The identifier should not change during the object's lifetime and once used should not be reused.

The mechanism for locating an object should be flexible enough to allow object migration. Three possible approaches to object migration are discussed below.

**Name Encoding:** The name encoding scheme encodes the location of an object within its object identifier. When an invocation is made, the system examines the appropriate field of the specified object identifier to determine the machine on which the object resides. Its main drawback is that the object cannot migrate, as this would require its name to change. To solve this problem, *forward location pointers* can be used. A forward location pointer is a reference used to indicate the new location of an object. Whenever an object migrates, a forward location pointer is created on the initial machine, to enable forwarding of invocation requests. However, this scheme cannot completely handle the problem of finding migrating objects since some pointers may be lost or may be unavailable due to failures.

**Distributed Name Server:** In this scheme, the system creates a group of name server objects that are maintained on several machines. These objects cooperate with one another so that they collectively contain up-to-date information about the location of every object in the system. There are two variations of this scheme. In the first variation, a name server maintains a complete collection of location information so each server can service any location request. In the second variation, partial information can be maintained by each server. If a location request cannot be serviced by one server, it is delegated to another.

**Cache/Broadcast:** In this scheme, a small cache is maintained on each workstation that records the last known locations of a number of recently referenced remote objects. When a client makes a remote invocation, the cache is examined to determine if it has an entry for the invoked object. If a location is found, the invocation request is sent to that machine. If

## CHAPTER 2. DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS

the object no longer resides at that machine, or if the location of the object is not recorded in the cache, a message is broadcast throughout the network requesting the current location of the object. Every machine receiving the broadcast examines the table of its local objects and sends a reply message to the originator of the broadcast if it finds the object. The entry in the cache is then updated. One problem with this scheme, is that broadcast requests involve overhead on all the machines even though only a single machine is directly involved with each location request.

### 2.6.2 System-Level Invocation Handling

The way a distributed, object-based system handles invocation requests depends entirely on the object model supported. Two schemes that are used are the message passing scheme and the direct invocation scheme.

**Message Passing:** A distributed, object-based programming system that provides the active object model typically supports the pure message passing scheme to handle object interactions. When a client makes an invocation on object, the parameters of the invocation are packaged into a request message. This message is then sent to a server process or a port associated with the invoked object. A server process in the invoked object accepts the message, unpacks the parameters, and performs the specified operation. When the operation completes, the result is packaged into a reply message, which is then sent back to the client. The main drawback of this scheme is that it involves excessive overhead for communications between objects residing on the same machine.

**Direct Invocation:** A distributed, object-based programming system that provides the passive object model typically supports the direct invocation scheme to handle object interactions. When a process invokes a server object that resides on the same workstation, the method invocation is then similar to a normal procedure call. An invocation on a remote object is similar to a remote procedure call. When the method of a remote object

## CHAPTER 2. DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS

is invoked, a message containing the parameters of the invocation is created and sent to the machine on which the server object resides. The machine that receives this message creates a worker process to execute on behalf of the original process. When the operation terminates, a message containing the results of the invocation is created and returned to the machine on which the original process resides. The worker process is then killed. This scheme incurs less overhead than the message passing scheme for local invocations since interactions between objects that reside on the same machine are relatively efficient. For remote invocations, this scheme has the added expense of creating and destroying worker processes.

### 2.6.3 Detecting Invocation Failures

There are two types of invocation failures.

**Existing fault.** An existing fault is defined as a failure that occurs before an invocation is started. The most common type of existing fault occurs when an invoked object cannot be located. These types of faults are relatively easy to detect and handle.

**Transient fault.** Transient fault occurs while an invocation request is being performed. They normally occur after a server object has accepted an invocation request but before the modifications made to it have been made permanent by the successful completion of the commit procedure. These faults are much more difficult to detect and handle because there are many different ways an invocation can fail. For example, the failure may be caused by the failure of the client object, the failure of the server object, or a network partition that separates a client from its server.

A distributed, object-based programming system should provide mechanisms for both client and server objects of the system to detect and recover from transient failures. Several simple failure detection schemes like time outs can be used. If the failure of an invocation is not detected by the client object, the client and the corresponding action may block and wait indefinitely. So a client must be able to detect invocation failures and initiate a

## **CHAPTER 2. DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS**

recovery procedure when one occurs. If the failure of an invocation is not detected by the server object, valuable system resources may be tied up unnecessarily.

### **2.7 The CORBA Architecture**

As defined by the Object Management Group(OMG) in [OMG91], the Object Request Broker (ORB) provides the mechanisms by which objects transparently make requests and receive responses. The ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems. The Common Object Request Broker Architecture (CORBA) is structured to allow integration of a wide variety of object systems.

#### **2.7.1 The Structure of ORB Interfaces**

Figure 2.3 shows a request being sent by a client to an object implementation. The client is the entity that wishes to perform an operation on the object and the object implementation is the code and data that actually implements the object. The ORB is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. The interface the client sees is completely independent of where the object is located, what programming language it is implemented in, or any other aspect which is not reflected in the object's interface.

Figure 2.4 shows the structure of an individual object request broker interfaces. The arrows indicate whether the ORB is called (a down arrow) or performs an up-call (an up arrow) across the interface. Interface Definition Language(IDL) is the language used to describe the interfaces that client objects call and object implementations provide. To make a request, the client can use the dynamic invocation interface (the same interface independent of the interface of the target object) or an IDL stub (the specific stub depending

CHAPTER 2. DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS

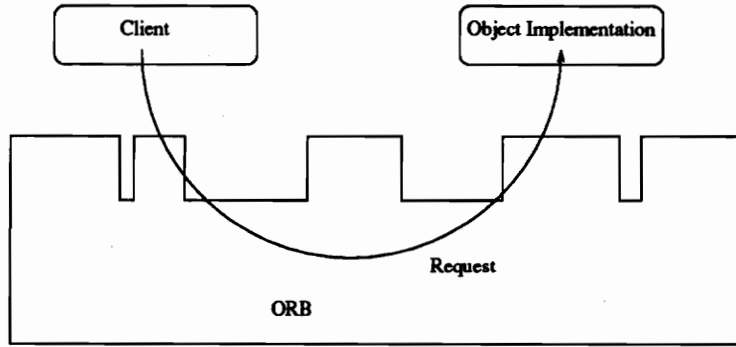


Figure 2.3: A request being sent through the Object Request Broker.

on the interface of the target object). Both techniques satisfy the same request semantics. So the receiver of the message cannot tell how the request was invoked. The client can also directly interact with the ORB. The object implementation receives a request as an up-call through the IDL generated skeleton. The object implementation may call the object adapter and the ORB while processing a request or at other times.

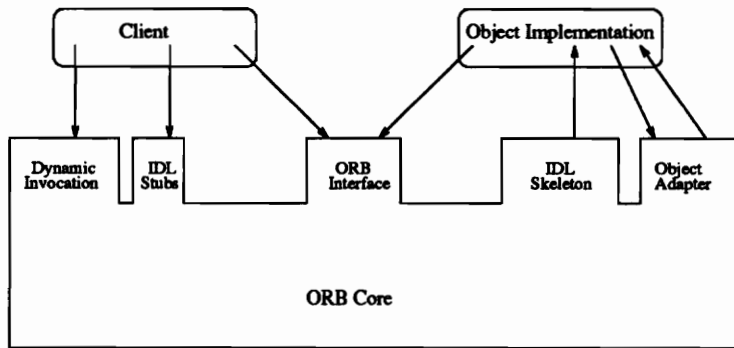


Figure 2.4: The structure of Object Request Broker interfaces.

Definitions of the interfaces to objects can be defined in two ways. Interfaces can be defined statically in an interface definition language (IDL). IDL defines the types of objects according to the operations that may be performed on them and the parameters to those operations. Alternatively, or in addition, interfaces can be added to an interface repository

## *CHAPTER 2. DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS*

service. This service represents the components of an interface as objects, permitting run-time access to these components. In any ORB implementation, the IDL and the interface repository have equivalent expressive power. When the client makes a request, the ORB locates the appropriate implementation code, transmits parameters and transfers control to the object implementation through an IDL skeleton. Skeletons are specific to the interface and the object adapter. When the request is complete, control and output values are returned to the client.

The object implementation interacts with the ORB in a variety of ways to establish its identity, to create new objects, and to obtain ORB-dependent services. The object implementation may choose which object adapter to use based on what kind of services the object implementation requires.

Object adapters are responsible for the following functions:

- generation and interpretation of object references,
- method invocation,
- security interactions,
- object and implementation activation and deactivation,
- mapping object references to the corresponding object implementations, and
- registration of implementations

It may be possible for a particular object adapter to delegate one or more of its responsibilities to the ORB core upon which it is constructed. ORB core is that part of the ORB that provides the basic representation of objects and communication of requests.

### **2.7.2 Interoperability**

The primary goal of the CORBA is to allow interoperation between different object systems and ORBs. For this we need a higher-level model that spans the differences among

## CHAPTER 2. DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS

different ORBs. In the case of the ORB, the higher-level model is the IDL-defined object-oriented invocation. Because IDL is defined in an ORB-independent way, it is possible for a particular request to pass through multiple ORBs, preserving the invocation semantics transparent to clients and implementations.

There are two ways to connect two ORBs.

- **Reference Embedding.** With reference embedding, an object in one ORB appears to be an object in a second ORB. An invocation on the object in the second ORB arrives at an implementation whose job is to perform an invocation in the first ORB.
- **Protocol Translation.** When two ORBs differ in their implementation details but have similar functionality, it may be possible to translate requests in one ORB to be requests in the other ORB by constructing a gateway.

The object model provides an organized presentation of object concepts and terminology. The OMG object model is *abstract* in that it is not directly realized by any particular technology.



# Chapter 3

## THE ACTOR MODEL

### 3.1 Description of the Actor Model

The actor model of computation was first developed by Hewitt, et al. [HBS73] and extended by Agha [Agha86].

In the Actor Model, the primary agents responsible for all activity are active objects called *actors*. An actor has a *mail address* (mailbox name) and a *script*, which is the representation of its behavior. The mail address is a pointer to a buffer which can store an unbounded number of messages that are received by this actor. The *behavior* corresponds to the actions undertaken by an actor in response to a message. Thus an actor can be modeled as shown in Figure 3.1.

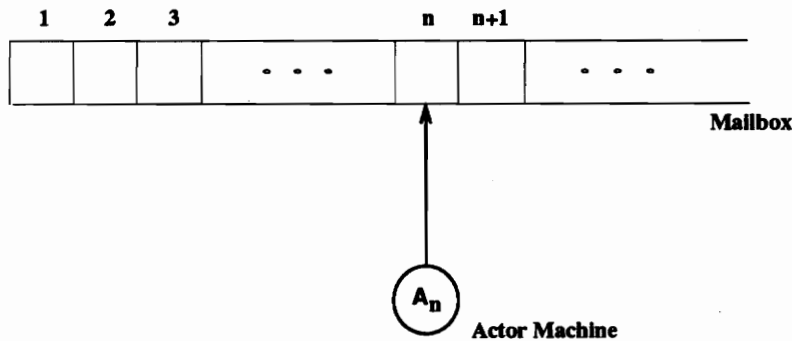


Figure 3.1: An abstract representation of an actor.

The identity of an actor is determined by its mailbox name, which is firmly decoupled from its state and behavior. Each instance of a behavior can process only one message. If

CHAPTER 3. THE ACTOR MODEL

the message queue is empty, when the actor is ready to receive the next message, the actor is blocked until a message arrives in the mail queue. The actions that can be taken by a behavior, as described by Figure 3.2, are the following:

- send one or more messages,
- create one or several new actors, and
- specify the replacement behavior that would process the next message

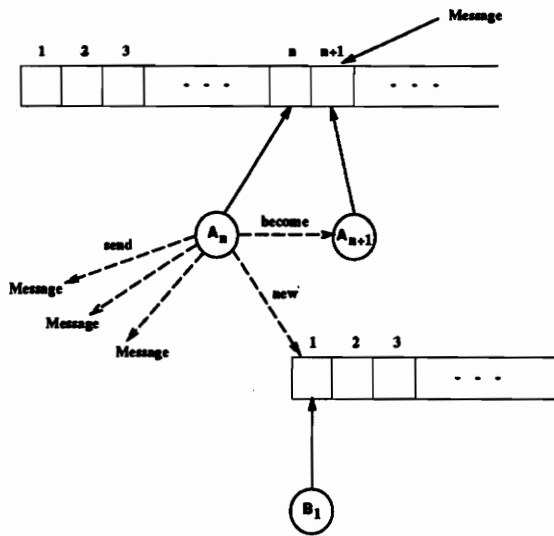


Figure 3.2: An abstract representation of transition.

An actor may communicate with a limited number of other actors, for which it knows the mailbox name, by sending messages. Those actors are said to be *acquaintances* of this actor. Communications are asynchronous. This means that when an actor sends a message to another actor, the sending actor may continue to do its own processing between the time it send a message and the time it receives a reply.

An actor may create new actors. The mail address of the newly created actors is known to the creating actor, which can disseminate the address to the other actors by sending messages containing the mail address of the newly created actor.

## CHAPTER 3. THE ACTOR MODEL

As a behavior can process only one message, a behavior must designate, using the operation `become`, a *replacement behavior*, which will process the next message.

Each actor has at least one thread of execution bound to it. Thus, the actor model supports concurrency, which can be observed at three levels. First, at the actor level, there may be several behaviors, each in a different actor, executing actions at the same time. Second, a single actor may have more than one behavior at one time. Third, there is also a concurrency at the instruction level. The instructions of a script may execute concurrently. However, this level of concurrency is not supported yet by most implementations of the actor model. Depending on when the `become` operation is performed, the replacement behavior may begin to process the next message concurrently with the creator of the replacement behavior.

Several concurrent languages and programming environments have chosen the actor model of computation as the underlying model of concurrency. Many of these languages incorporate only part of the features of the actor model either because of the choice of the base language or the requirements of the intended application domain. ACT++, which is an actor-based language, is described below.

### 3.2 ACT++: An Implementation of the Actor Model

ACT++ is an implementation of the actor model in C++. ACT++ is a class library written in C++ that implements the abstraction of the actor model, with slight modifications. Instruction-level fine-grain concurrency is not available in ACT++. Thus, in ACT++, a behavior in execution is a lightweight thread of control. The main features of ACT++ are described below.

An ACT++ program does not consist only of actors. Objects of all basic types available in C++ along with those of any user-defined type coexist with actors in an ACT++ program. Special classes are available to implement actors and the asynchronous method invocation associated with them. Each of the three primary notions involved in program-

## CHAPTER 3. THE ACTOR MODEL

ming with actors, namely actors, behaviors and messages, are instantiations of predefined classes in ACT++.

There are two types of messages: request messages and reply messages. A request message corresponds to a method invocation, while a reply message contains the result of a method invocation. Request messages are sent to the mailbox of an actor. The execution of an actor's current behavior is initiated by the availability of a request message. Request messages are buffered in the message queue of the actor and are processed in a first-in first-out order. A request message contains the name of the method to be invoked and its parameters. A request message can contain one or more Cbox names if a result is to be returned by an invocation. A Cbox can be considered as a mailbox for reply messages. A reply message contains the result of an operation. Reply messages are addressed to a Cbox. To obtain the value from the Cbox, the actor uses the overloading of the type cast operation. This operation is a blocking operation, which means that the behavior will block until it receives a reply.

### 3.3 A Distributed Actor System

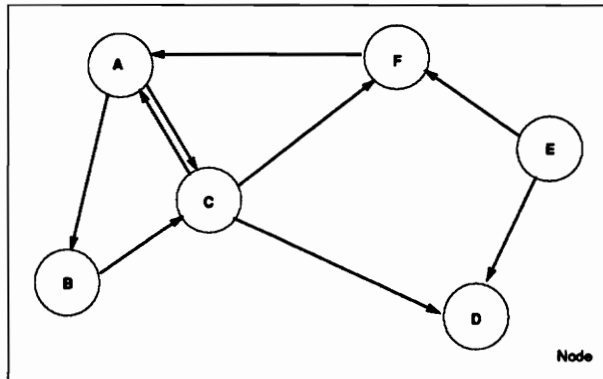


Figure 3.3: A centralized actor system.

An actor system is a collection of actors that interact with each other. In a centralized environment, an actor system can be depicted by a graph, as shown in Figure 3.3. Actors are

### *CHAPTER 3. THE ACTOR MODEL*

represented by the nodes, and the possibility for an actor to send a message to another actor by a directed arc. In a shared memory environment, an acquaintance is typically a pointer to an actor. However, in the case of a distributed actor system, acquaintances cannot be implemented with pointers. In a distributed system, the scope of a pointer is limited to one node of the distributed system. An actor identifier scheme must be chosen and support is needed to generate and interpret actor references. In a heterogeneous environment, a mechanism to encode the messages in an architecture independent format is necessary as different machines may implement basic data types differently. The sender then encodes the message in this format. The encoded message is sent through the network to the receiver, which decodes the message. Support for dynamic loading and unloading of an actor implementation is also necessary, as an actor implementation may be present on several nodes, but used on only one node. The duplication of implementation may be useful to ensure the availability of a service in the case of a workstation failure, or to allow load balancing between different nodes. Finally actor implementations must be registered to allow the dynamic linking of the object file which is defining an actor. Figure 3.4 represents the previous system, once it has been distributed. It must be noted that the message encoding and transmission is not represented in this figure.

Local invocations and actor creations are still processed in the same way as for the centralized system. On each node, a server is present which handles all the messages arriving at a node. The server decodes a message, loads the object implementation in memory if needed, and then invokes the constructor or method. An implementation the kernel is presented in the next chapter.

CHAPTER 3. THE ACTOR MODEL

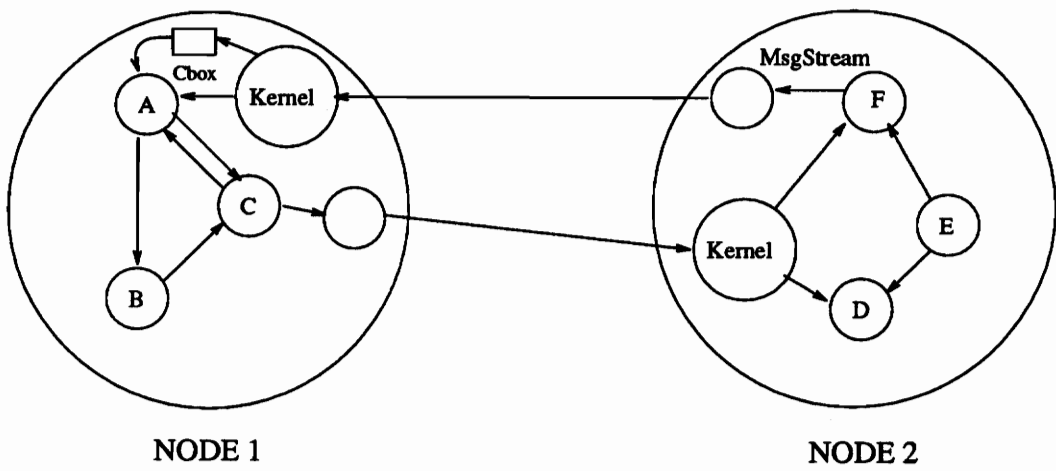


Figure 3.4: A distributed actor system.

# Chapter 4

## AN EXECUTION ENVIRONMENT FOR ACT++

### 4.1 Overview of the Environment

The environment consists of the following components as shown in Figure 4.1:

- a **server** for handling the incoming messages,
- a **list manager** to handle the communications with the server,
- a mechanism to dynamically load and unload object implementations,
- a mechanism to statically load object implementations for machines that do not support run-time loading,
- a directory service that locates object implementations in memory, called the **Class Directory**,
- a directory service that locates an actor in memory, called the **Actor Directory**,
- a directory service that locates an input/output actor, called the **Message Stream Directory**,
- a directory service that locates Cboxes in memory, called the **Cbox Directory**, and
- a directory service to locate various methods of a particular class in memory, called the **Method List Directory**.

CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

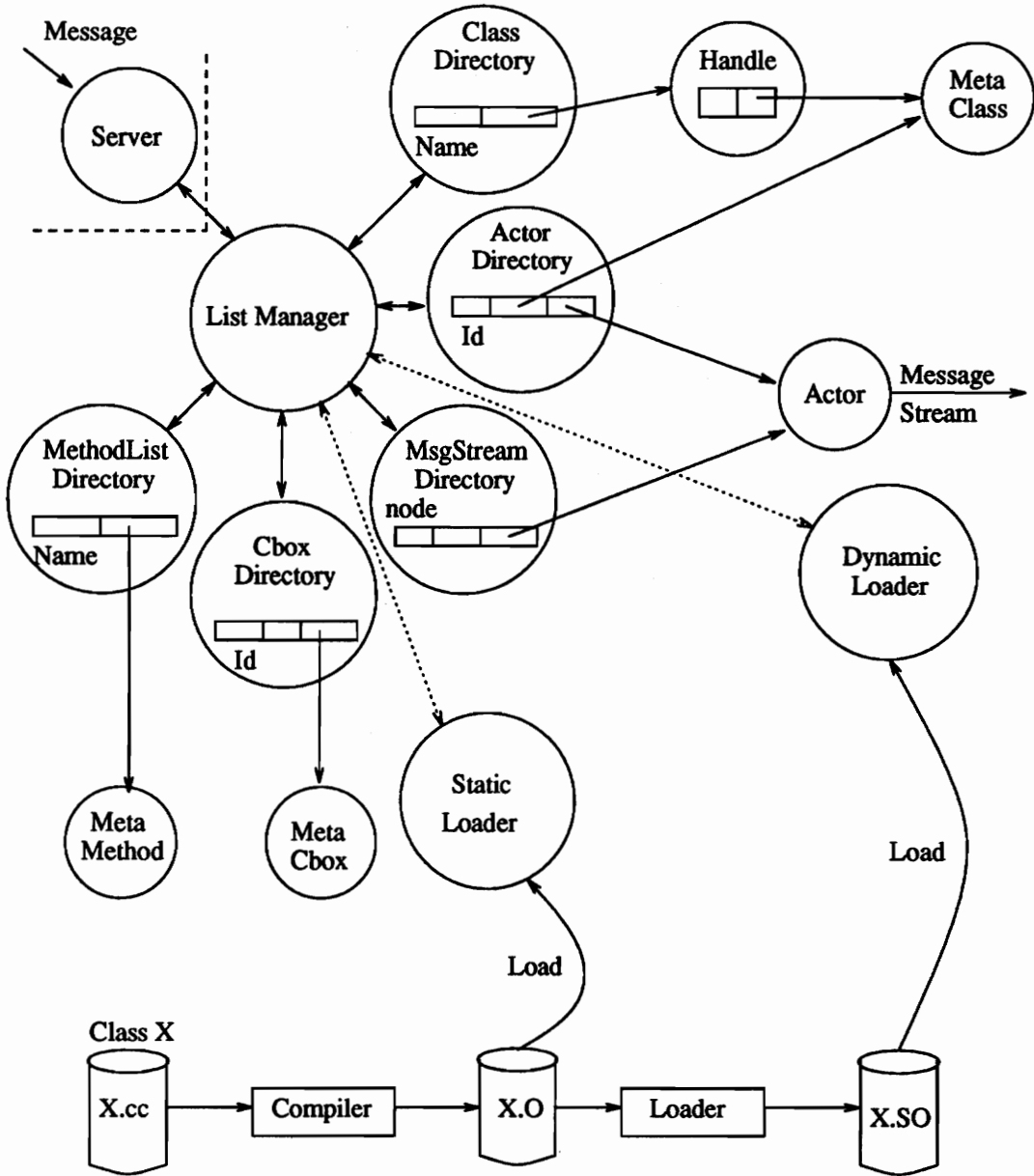


Figure 4.1: An overview of the environment.



## CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

### 4.2 Details of the Current Implementation

The design and implementation of the server itself is outside the scope of this project. This section discusses in detail only the communications the server performs with the other components.

When a message is received by the **server**, it is first decoded to know whether the message is one of the following messages.

- *Constructor Message*. A request for the creation of the Actor
- *Method Invocation Message*. A request for an invocation of the method of an Actor
- *Reply Message*. A message to return the result of an invocation
- *Destructor Message*. A request for the destruction of an Actor

The first two messages and the last message are relevant to an actor, while the third message is relevant to a Cbox, which is the mechanism used by the distributed ACT++ system to return the result of an invocation. The response of the server to each of these messages is discussed in Section 4.2.1.

#### 4.2.1 The Communications Interface with the Server

##### Actor Constructor Message

The format of the constructor message is shown in Figure 4.2

Class Name	Constructor Arguments	Reply Point
------------	-----------------------	-------------

Figure 4.2: The actor constructor message format.

When the server receives an actor constructor message, the following sequence of events occur as shown in Figure 4.3.

CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

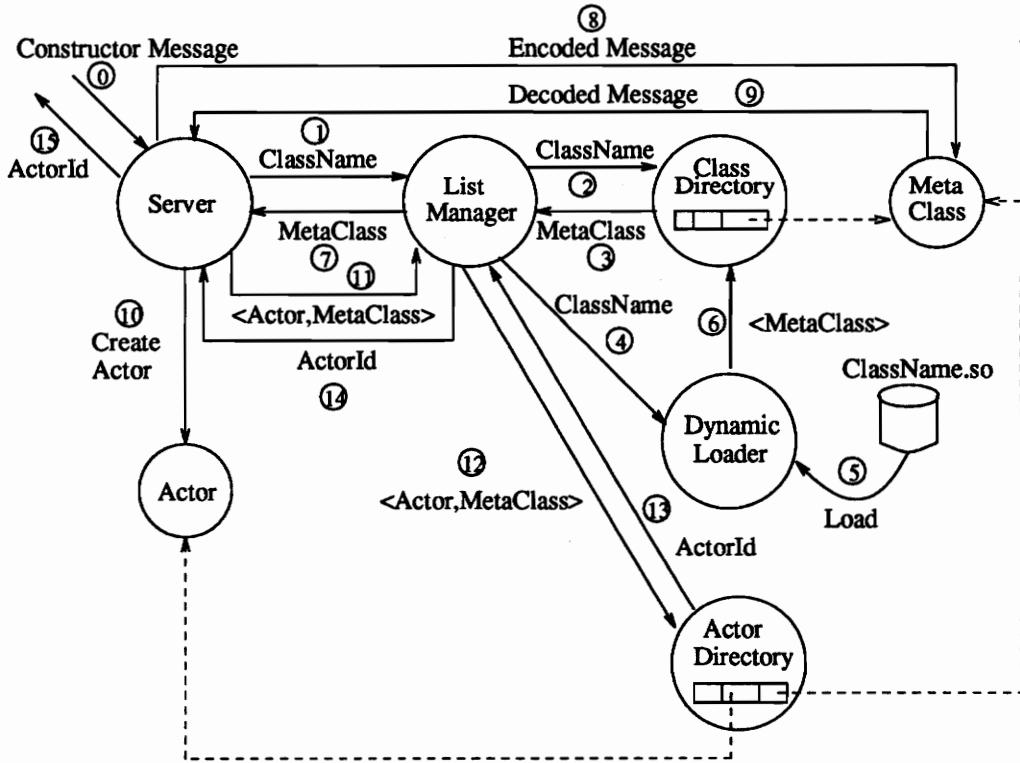


Figure 4.3: The actor construction process.

1. The server first sends the class name to the list manager service.
2. The list manager sends the class name to the class directory service.
3. If the object implementation for that class is available in memory then the class directory returns a pointer to the meta class of the object implementation. If it is not available in memory, it returns a null pointer to the list manager. The purpose of the meta class is to handle the decoding of the method invocation arguments, including invocation of constructor methods, for all objects of that class. In the case of overloaded and virtual functions, the meta class has the responsibility of selecting the right function to be called.
4. If the list manager gets a null pointer from the class directory service, it sends the class

## CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

name to the dynamic loader to dynamically load the shared object code of an instance of the class. If the list manager gets a non-null pointer from the class directory, the list manager performs the actions from step 7.

5. The dynamic loader performs dynamic linking.
6. The dynamic loader makes an entry in the class directory for future references and returns a pointer to the meta class of the object implementation.
7. The server gets the pointer to the meta class from the list manager.
8. The server sends the remaining part of the message to the meta class to be decoded. If the meta class is not in memory an error message is entered in the error log file.
9. The meta class object returns the decoded message to the server.
10. The server uses this decoded message to construct the behavior of the actor and then the actor itself.
11. The server then sends the pointer to the actor and the pointer to the meta class to the list manager.
12. The list manager sends these pointers to the actor directory to add an entry for future references.
13. The actor directory service returns an actor identifier. .
14. The list manager gives the actor identifier to the server.
15. The server then returns the actor identifier to the initiator of the call, by sending a reply message.

### **Actor Method Invocation Message**

The message received for a method invocation request, has the structure shown in Figure 4.4.

CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

ActorId	Method Name	Method Arguments
---------	-------------	------------------

Figure 4.4: The actor method invocation message format.

When the server receives an actor method invocation message, the following sequence of events occur as shown in Figure 4.5.

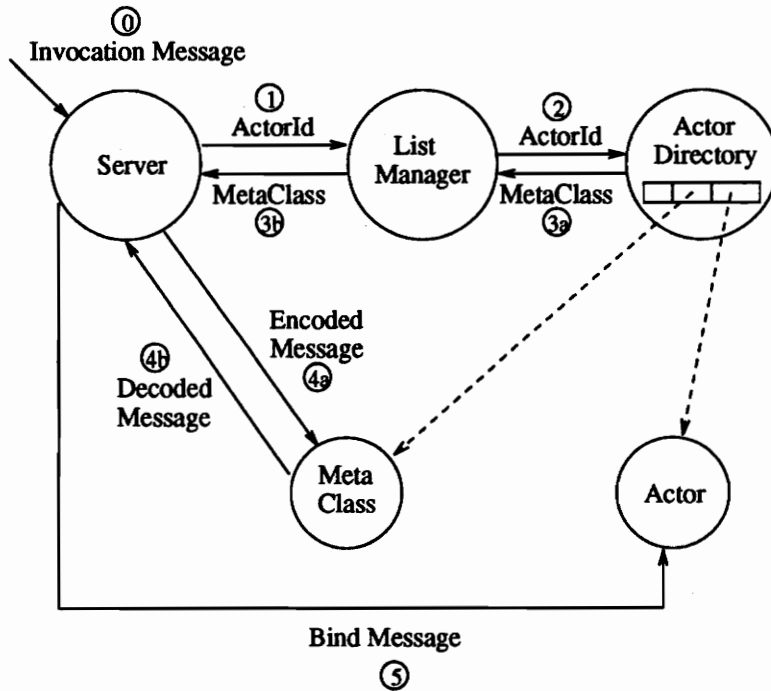


Figure 4.5: The actor method invocation process.

1. The server first sends an actor identifier to the list manager service.
2. The list manager sends the actor identifier to the actor directory service.
3. The actor directory service returns a pointer to the actor and a pointer to the meta class of the object implementation of an instance of the class that was used to construct the actor.

**CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++**

4. The server sends the remaining part of the encoded message to the meta class to be decoded. The meta class returns the decoded message.
5. The server then sends the decoded message to the actor.

**Reply Message**

The format of the reply message is shown in Figure 4.6. In the case of a reply message, the message is decoded to obtain the CboxId. The remaining data is forwarded to the Cbox using the Cbox directory as shown in Figure 4.7.

CboxId	Method Invocation Result
--------	--------------------------

Figure 4.6: The reply message format.

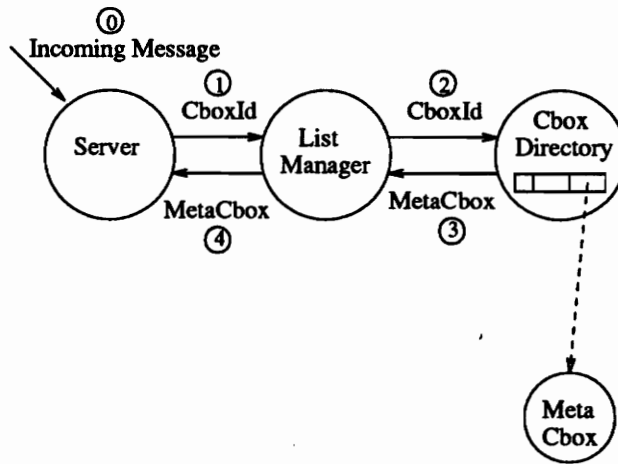


Figure 4.7: The reply process using cbox directory.

When the server receives a reply message, the following sequence of events occur as shown in Figure /refpict15.

- The server sends the cbox identifier to the list manager.

## CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

- The list manager sends the cbox identifier to the cbox directory.
- The cbox directory sends a pointer to the meta cbox to the list manager. If the meta cbox is not available in memory the cbox directory sends a null pointer to the list manager.
- The list manager sends the pointer to the meta cbox to the server. The server uses the meta cbox to place the result of the method invocation.

### Destructor Message

In the case of a destructor message, the destructor is called using the same mechanisms as for a regular method invocation. In addition, the server sends a request to the list manager to remove the entry corresponding the actor being destructed. The list manager sends this message to the actor directory. The destructor message is similar to a method invocation message, except that a destructor has no arguments, as shown in Figure 4.8.

When the server receives an actor destructor message, the following sequence of events

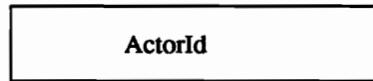


Figure 4.8: The actor destructor message format.

occur as shown in Figure 4.9.

- The server sends the actor identifier to the list manager.
- The list manager sends the actor identifier to the actor directory. The actor directory deletes the entry corresponding to the actor identifier and frees up the memory.

### 4.2.2 The List Manager Class

The list manager is responsible for handling the communications with the server and for providing the directory services to the server in the distributed actor system. The

## CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

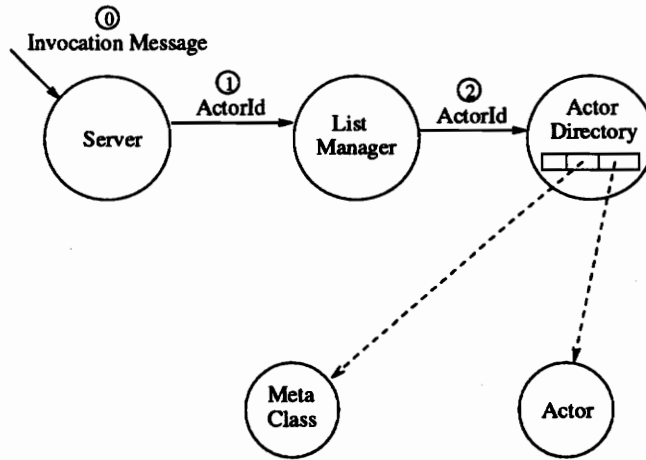


Figure 4.9: The actor destructor process.

ListManager class is defined as follows.

```
class ListManager {  
    ActorList    actorList;  
    ClassList    classList;  
    CboxList     cboxList;  
    MsgStreamList msgStreamList;  
    MethodList   methodList;  
  
public:  
    ListManager();  
    void    staticLink(String className, MetaClass* ptrMetaClass);  
    MetaClass* findClass(String className);  
    MetaClass* findClass(int actorId);  
    MetaCbox* findCbox(int cboxId);  
    Actor* findActor(int actorId);  
    void deleteActor(int actorId);  
    void deleteCbox(int cboxId);  
};
```

## CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

```
void    deleteClass(String className);
int     enter(MetaCbox* ptrMetaCbox);
int     enter(Actor* ptrActor, MetaClass* ptrMetaClass);
void    CboxRef(int cboxId);
}
```

The `staticLink` method is invoked by the static loading mechanism discussed in Section 4.2.7. The static loading mechanism relies on this method to make entries of the statically loaded object implementations in the class directory service. The list manager is also responsible for the dynamic loading of the object implementations when they are not available in memory. A mechanism to dynamically load object implementations is discussed in Section 4.2.6. When the server receives an actor constructor message, the list manager queries the class directory service about the availability of the meta class of the object implementation. If it is available, it returns a pointer to the meta class. If it is not available, the list manager performs dynamic loading and makes an entry in the class directory. The list manager can also perform dynamic unloading of object implementations when they are no longer needed.

### 4.2.3 The Handle Class

The `Handle` class is primarily responsible for the dynamic loading and unloading of object implementations. The `Handle` class is defined as follows.

```
class Handle {
    void*      hdl;
    static int  IfFileInSearchPath(char* path, char* fileName);
    static char* LocateSharedObject(char* fileName);
public:
    MetaClass* metaClass;
    Handle();
}
```



## CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

```
    MetaClass*   link(const char* className);  
    void        unlink();  
}
```

The `link` method in the `Handle` class specifically does dynamic loading of the object implementation of an instance of a class with the name `className`. It reads the environment variable `LD_LIBRARY_PATH` set by the user to search for the file containing the shared object implementation of the class. For every dynamically added object implementation, an instance of the class `Handle` is created. The method `unlink` is used by the list manager for dynamic unloading of object implementations.

### 4.2.4 The Map Class

The Map template class is used by all the directory services in the distributed ACT++ system. It defines a template for an associative array, also called Map or dictionary. A Map keeps pairs of values. The first component is called the key and the second its value. The main purpose of a Map is to retrieve the value corresponding to a key.

Two classes are defined: `Map` and `Mapiter`. The class `Map` is the class defining the table. The class `Mapiter` is a friend class of `Map` and is a class providing iteration functions to examine the table. Several iterators of the class `Mapiter` can be instantiated, thus allowing concurrent interrogation of a table.

Part of the implementation of this Map class can be found, along with some comments in [STR91].

### 4.2.5 The Directory Services

The directory services are implemented using the Map class described in Section 4.2.4.

## CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

### The Class Directory

The key for the class directory is `class name`. The value, `ClassListValType`, is a structure which is defined as follows.

```
typedef struct ClassListValType {
    Handle* handle;
} ClassListValType;
```

The class `ClassList` is defined as follows.

```
class ClassList {
    Map<String, ClassListValType> Directory;
public:
    ClassList();
    void        enter(String className, Handle* hdl);
    void        remove(String className);
    int         IfPresent(String className);
    ClassListValType lookup(String className);
}
```

The method `enter` is used for adding a new entry in the class directory. The method `lookup` is used to retrieve the information in the directory based on `className`, the name of the class. The method `remove` is used for deleting an entry in the class directory service. The Method `IfPresent` is used by the list manager to find whether the object implementation of a particular class with the class name, `className` is available in memory.

### The Actor Directory

The key for the actor directory is an Actor identifier, `actorId` on this node. The value, `ActorListValType`, is a structure which is defined as follows.

```
typedef struct ActorListValType {
```

## CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

```
        MetaClass* meta;
        Actor* act;
    } ActorListValType;
```

The class `ActorList` is defined as follows.

```
class ActorList {
    int NextActorId;
    Map<String, ActorListValType> Directory;
public:
    ActorList(int actorId=0);
    int enter(MetaClass* ptrMetaClass, Actor* ptrActor);
    void remove(int actorId);
    int IfPresent(int actorId);
    ActorListValType lookup(int actorId);
}
```

On each machine, a local directory keeps track of all the actors that have been created locally. To accomplish this the actor directory service assigns an integer id to each actor created on the local system. The actor directory keeps a counter which represents the identifier of the next actor to be created. When it receives a request to add an entry in the database, the value of the counter is bound to this actor and returned to the server. The counter is then incremented. The method `lookup` returns the structure `ActorListValType` corresponding to the actor identifier given. The structure `ActorListValType` contains a pointer to the actor in memory, as well as a pointer to the meta class that corresponds to the class from which the behavior of this actor has been created.

### The Message Stream Directory

The key for the message stream directory is the logical name of a node participating in the distributed ACT++ system. The value, `MsgStreamListValType`, is a structure which

## CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

is defined as follows.

```
typedef struct MsgStreamListValType {
    Actor*      act;
    long        IDNumber;
} MsgStreamListValType;
```

The class `MsgStreamList` is defined as follows.

```
class MsgStreamList {
    Map<String, MsgStreamListValType> Directory;
public:
    MsgStreamList();
    void          enter(String nodeName, Actor* ptrActor, long number);
    void          remove(String nodeName);
    void          remove(long interfaceActorId);
    Actor*        lookup(long interfaceActorId);
    MsgStreamListValType* lookup(String nodeName);
}
```

Input/output in distributed ACT++ has been implemented as an actor operation. This means that instead of issuing I/O requests directly to sockets the program must direct such requests to an actor that is responsible for that socket. Actors which handle I/O to sockets are called *interface actors*. The message stream list directory keeps track of all the interface actors that have been created locally. It provides the `lookup` facility based on both the logical name of the node and the interface actor identifier, which is a long integer. All messages to the other nodes in the distributed actor system are sent through the message stream using interface actors.

## CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

### The Cbox Directory

The key for the Cbox directory is a cbox identifier, `CboxId` on this node. The value, `CboxListValType`, is a structure which is defined as follows.

```
typedef struct CboxListValType {
    int          refCount;
    MetaCbox*   metaCbox;

    CboxListValType(MetaCbox* mcb)
        : refCount(0), metaCbox(mcb) {}
    CboxListValType()                {}
} CboxListValType;
```

The class `CboxList` is defined as follows:

```
class CboxList {
    int CboxId;
    Map<String, CboxListValType> Directory;
public:
    CboxList(int cboxId=0);
    int          enter(MetaCbox* ptrMetaCbox);
    void         remove(int cboxId);
    MetaCbox*   lookup(int cboxId);
    int         IfPresent(int cboxId);
    void         CboxRef(int cboxId);
}
```

The method `CboxRef` is used to increment the reference count of a cbox. The method `remove` removes an entry in the cbox directory only when the reference count corresponding to that cbox is zero. The reference counting scheme is not used in the current version of the distributed actor system.

## CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

### The MethodList Directory

The key for the method list directory is the name of the class. The value, MethodListValType, is a structure which is defined as follows.

```
typedef struct MethodListValType {
    MetaMethod* metaMethod;
} MethodListValType;
```

The class MethodList is defined as follows.

```
class MethodList {
    Map<String, MethodListValType> Directory;
public:
    MethodList();
    void enter(String className, MetaMethod* ptrMetaMethod);
    void remove(String className);
    int IfPresent(String className);
    MetaMethod* lookup(String className);
}
```

The method list directory keeps track of all the available methods of a particular class. The information about all the methods of a class is hidden inside the meta method of a class. The lookup method returns a pointer to the meta method of an object implementation of a class.

### 4.2.6 Dynamic Loading Mechanism in Distributed ACT++

The list manager performs the dynamic loading of the shared object implementation of a class when it is not available in memory. We can create shared libraries from object files or from archive libraries. The mechanism discussed in this section uses the environment variable LD\_LIBRARY\_PATH as the search path for finding the shared object implementation

## CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

of a class. The method `link` in the `Handle` class does the dynamic loading. It is defined as follows.

```
MetaClass* Handle::link(const char* className) {
    char* fileName = new char[strlen(className)+4];
    (void)sprintf(fileName,"%s.so",className);
    char* fileLoc = LocateSharedObject(fileName);
    if(!fileLoc) {
        cout << fileName << "Not Found \n";
        delete[] fileName;
        return (MetaClass *)0; }
    char* filePath = new char[strlen(fileName)+strlen(fileLoc)+2];
    strcpy(filePath,fileLoc);
    strcat(filePath,"/");
    strcat(filePath,fileName);
    delete[] fileName;
    delete[] fileLoc;
    hdl = dlopen(filePath, RTLD_LAZY);
    if(!BAD(hdl)) {
        char* DldFnName = new char[strlen(className)+12];
        (void)sprintf(DldFnName,"%s_Maker__Fv",className);
        MetaClass* (*Dldfp)();
        Dldfp = dlsym(hdl, DldFnName);
        if(!BAD(Dldfp)) {
            metaClass = (*Dldfp)();
            delete[] DldFnName;
            delete[] filePath;
            return metaClass; }
        else {
```

## CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

```
    cout << " dlsym error " << dlerror() << "\n";
    delete[] DldFnName;
    delete[] filePath;
    return (MetaClass *)0; }
} else {
    cout << "dlopen error " << dlerror() << "\n";
    delete[] filePath;
    return 0; }
}
```

The method `Handle::LocateSharedObject` finds the path of the shared object implementation of the class using the environment variable `LD_LIBRARY_PATH` set by the user. The object code of a class with the name `ClassName` is considered to be located in a shared object file with the name `ClassName.so`.

To dynamically load an object implementation of a class we need a mechanism to create an instance of a class. This is accomplished using a *classMaker* function which returns an instance of a class. For example, to define a stack class with the `stackops` behavior, the following line is added by the user to the file containing the definitions of the stack class.

```
IMPLEMENTS(stack, stackops)
```

When the file containing the class definitions passes through a stub generator, it adds the following source code.

```
MetaClass* _stack_Maker() {return new stack<stackops>;}
```

This is simply obtained by expanding using the following macro definition.

```
#define IMPLEMENTS(Interface,Operations)\
    MetaClass* _ ## Interface ## _Maker() {return new Interface<Operations>;}
```

The function generated using this macro definition returns an instance of a class when called. The g++ compiler does mangling to implement the feature of overloaded functions.



## CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

From the g++ name mangling scheme, we can easily determine the mangled name of the above function. The mangled name of the above function is `_Interface_Maker_Fv`. We get a handle to this function using the `dlsym` function call. Once a handle is obtained, this function is called whenever an instance of a class *Interface* with the behavior *Operations* is needed. The `dlsym` function call returns a pointer to a meta class. The list manager keeps track of dynamically loaded object implementations of a class.

The method `Handle::unlink` does dynamic unloading of object implementations. It is defined as follows.

```
void Handle::unlink() {
    if(hdl) dlclose(hdl);
}
```

The variable `hdl` is the handle returned by the `dlopen` function call during the dynamic loading of the object code. The function `dlclose` disassociates a shared object previously opened by `dlopen` from the current process. Once an object has been closed using `dlclose` its symbols are no longer available for reference.

### 4.2.7 Static Loading Mechanism in Distributed ACT++

The static loading mechanism is primarily developed to preload the object code of the most frequently used classes for run time efficiency. This mechanism can also be used by machines that do not support the dynamic loading of shared object implementations. As soon as the server begins execution, the static linking mechanism should be able to create entries in the class directory for all the classes statically linked with the server code. It is accomplished using the combination of macro expansions and constructor functions. A template `staticLink` is used to provide the constructor function to make entries in the class directory. It is defined as follows.

```
extern ListManager listMgr;
template<class T> class staticLink {
```

## CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

```
public:
    staticLink(char* className) {
        MetaClass* m = (MetaClass *) new T();
        listMgr.staticLink((String)name,m); }
}
```

The `listMgr` is a global list manager. The constructor function of the class `staticLink` calls the method `ListManager::staticLink` which makes an entry in the class directory service for future references. Macros are used to create an instance of the class `staticLink`. To statically load an object implementation the user writes the following line in the source file containing the class definition.

```
STATIC_LINK(Interface, Operations)
```

The stub generator uses the following macro definition for expansion.

```
#define STATIC_LINK(Interface, Operations)\
    staticLink< Interface<Operations> > staticLink_ ## Interface ## (\
    # Interface ##);
```

Using the above macro definition, the stub generator produces the following source code:

```
staticLink<Interface<Operations>> staticLink_Interface("Interface");
```

When the server begins execution, the above piece of code generates a call to the constructor function of `staticLink` which updates the class directory service. In the source code of the server care should be taken to place the declaration of the `listMgr` before the statement `STATIC_LINK(Interface,Operations)` to prevent the execution of a method in a class before it is actually created.

The communication between the client and the server is tested using several simple object implementations. The details are presented in the next section.

### 4.2.8 Functional Testing

For testing purposes, a specialized testing environment was created in which the communication between the server and the client is actually simulated by the use of a single pipe. The only difference between this test environment and the real system is that the server is reading the incoming messages from the standard input instead of a socket. The testing environment is shown in the Figure 4.10. The messages generated by the client are sent through the one-way pipe and received at the other end of the pipe by the server. The environment variable `LD_LIBRARY_PATH` specifies the search path used by the dynamic loader to locate the object implementations. The tests are done by changing either the type of messages, the number of messages, the environment variables and the location of the object files in the file system.

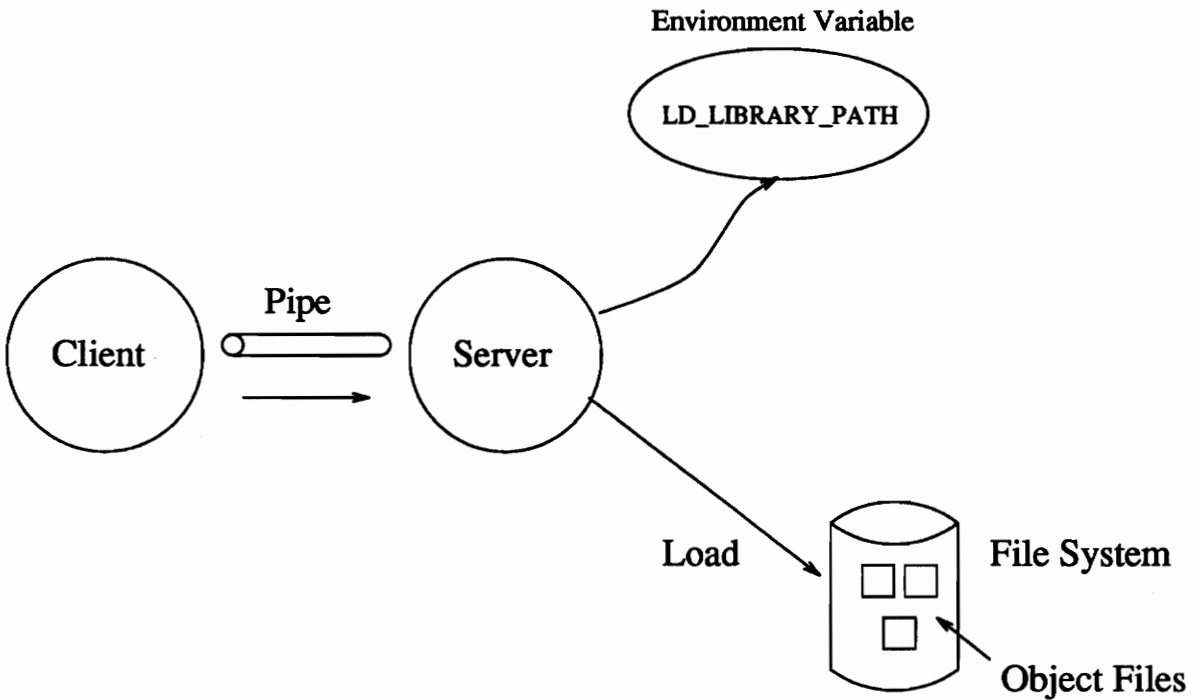


Figure 4.10: The testing environment.

The testing of the dynamic loading mechanism is done on a Sun Sparc workstation

## CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

running Solaris 2.3. Solaris 2.3 provides support for dynamic linking of shared object files. The testing of static loading of object classes is done on DEC Ultrix 4.3 and on Sun Solaris 2.3. There is no support for dynamic linking on the DEC Ultrix 4.3 platform.

The tests that were conducted are described below and summarized in Figure 4.11.

Elements	Dynamic Loading	Static Loading	Combined	Others
List Manager	✓	✓	✓	
Class Directory	✓	✓	✓	
Actor Directory	✓	✓	✓	
Message Stream Directory				✓
Cbox Directory				✓
Method List Directory				✓

Figure 4.11: Testing the components of the environment

### Dynamic Loading

The environment variable `LD_LIBRARY_PATH` specifies the search path for finding the shared object implementation of a class. The method `link` in the `Handle` class does the dynamic linking. For example, we can set the environment variable using the following command.

```
setenv LD_LIBRARY_PATH ./usr/local/lib:/home/vykunta/lib
```

## CHAPTER 4. AN EXECUTION ENVIRONMENT FOR ACT++

The dynamic loader initially searches the current working directory for shared object implementations. If it does not find the object implementations it searches `/usr/local/lib` for the object code. If it does not find the object code there, it continues searching in `/home/vykunta/lib`. If it fails to locate the object code in the entire path specified by the environment variable, it generates an error message.

By setting different paths for `LD_LIBRARY_PATH`, the robustness of the dynamic loading mechanism to locate the shared object implementations is tested. For example, in testing two versions of the `stack` objects were used. One `stack` object implementation is in `/home/vykunta/version1`. Second `stack` object implementation is in `/home/vykunta/version2`. The environment variable is set using the following command.

```
setenv LD_LIBRARY_PATH /home/vykunta/version2:/home/vykunta/version1
```

When the server side receives a `stack` actor constructor message, the dynamic loader searches the `LD_LIBRARY_PATH` and loads the `stack` object code in `/home/vykunta/version2`. This test confirmed that the dynamic loader loads the shared object implementations of a class when it is not available in memory using the environment variable `LD_LIBRARY_PATH`.

### Static Loading

Static loading of object classes is tested on DEC Ultrix 4.3 and Sun Solaris 2.3 platforms. As soon as the server begins execution, the static loading mechanism creates entries in the class directory for all the classes statically linked with the server code. The remote client that invokes a method on the remote object doesn't know whether the object is dynamically loaded or statically loaded. For testing purposes, a `stack` object is statically loaded. When the server receives a `stack` actor constructor message, the `stack` actor is constructed using the pre loaded `stack` object implementation. This test confirmed that the static loading mechanism pre loads the object implementations and creates entries in the class directory for future invocations of the methods on the object.

### Combined Tests

For testing purposes, I have created two versions of stack objects. The two versions of stack objects differ in the type of messages printed when the pop method is invoked. One version of the stack object is statically loaded. Second version of the stack implementation is dynamically loaded. The test consists of sending the stack constructor messages for both the versions of the stack object to the server. For each version of the stack object one hundred method invocation messages are sent from the client to the server. When the server receives the stack constructor message for the stack object that is not available in memory, it performs dynamic loading and loads the object implementation of the stack. When the server receives the method invocation message, it invokes the method of that particular stack object. This test confirmed that in a distributed actor system some classes can be statically loaded and some classes can be dynamically loaded.

### Other Tests

Some of the components of the environment developed in this project were tested by others. The message stream directory, the cbox directory, and the method list directory are tested by others who are involved in the design and development of distributed ACT++ 3.0.

The dynamic loading mechanism has the support for error handling. Three error conditions are taken care of in the design of the dynamic loading mechanism. One possibility is that the dynamic linker fails to locate the shared object implementations of the class. Second possibility is that the dynamic linking of the object code fails. Third possibility is that the mechanism to obtain a handle to the *ClassMaker* function fails. In all the cases, error messages are entered in the appropriate log files. The loading mechanisms developed in this project are subjected to *stress testing*. Four hundred messages are sent from the client to the server side for remote invocations.

## Chapter 5

# CONCLUSIONS AND FUTURE WORK

The system described in this report performs the following functions.

- It dynamically loads an object implementation of a class when it is needed and it is not available in memory.
- It statically loads the object implementations that are used most frequently in the system. Machines that do not support the dynamic linking of object code can use this facility.
- Provides directory services to locate classes and their instances in the distributed actor system.

The dynamic loading facility makes use of the name mangling scheme adopted by the g++ compiler. If the program is compiled with a different C++ compiler, then the method `Handle::link` must be modified to obtain the handle to the *InterfaceMaker* function using the `dlsym` function call. The other components are completely portable.

The environment created does not support the property of location transparency. The client needs to know where the object implementation resides in order to invoke the constructor of this object. In fact, one has to indicate the machine on which the object is to be created. This may happen for two reasons.

- In the absence of an object migration capability, it may be desirable to create an object on the machine that has the lowest load.
- The replication of a service will ensure availability of the service in the case of a workstation failure.

## *CHAPTER 5. CONCLUSIONS AND FUTURE WORK*

The features of availability and load balancing can be added to the distributed ACT++ system by creating a distributed global database of the available classes in the system.

The distributed ACT++ system can also be modified to support multiple languages.



## REFERENCES

- [Ada9X] Ada 9X Mapping/Revision Team. *Programming language Ada - language and standard libraries*, Draft 4.0, September 1993.
- [Agha86] Gul A. Agha. *ACTORS: A model of concurrent computation in Distributed Systems*. The MIT Press, Cambridge, MA, 1986.
- [Barnes] J.G.P. Barnes. *Programming in Ada* Addison-Wesley, CA, second edition.
- [CC91] Roger S. Chin and Samuel T. Chanson. Distributed Object-Based Programming Systems. *ACM Computing Surveys*, 23(1):91-124, March 1991.
- [Gold] Adele Goldberg. *Smalltalk-80 The Interactive Programming Environment*. Addison-Wesley, CA.
- [HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd IJCAI*, pages 235-245, 1973.
- [NEL] Greg Nelson. *Systems Programming with Modula-3*. Prentice Hall, NJ, 1991.
- [OMG91] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, Draft 10, December 1991. Document Number 91.12.1.
- [STR91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, CA, second edition, 1991.
- [WEG87] Peter Wegner. Dimensions of object-based language design. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems Languages and Applications*, pages 168-182, December 1987. ACM.
- [Wirth] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, Second Edition.