Communication Infratructure for a Distributed Actor System

by

Rajiv Gandhi

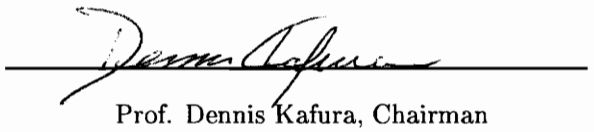
Project Report submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

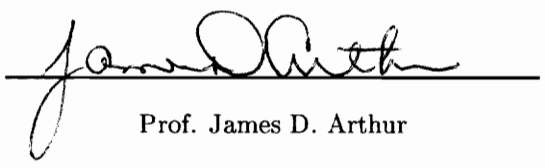
in

Computer Science

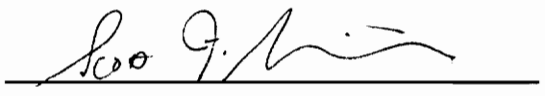
APPROVED:



Prof. Dennis Kafura, Chairman



Prof. James D. Arthur



Prof. Scott F. Midkiff

June, 1994

Blacksburg, Virginia

Communication Infratructure for a Distributed Actor System

by

Rajiv Gandhi

Committee Chairman: Prof. Dennis Kafura

Department of Computer Science

(ABSTRACT)

The goal of this project was to develop part of the environment that would allow the creation of distributed applications in ACT++. ACT++ is a programming framework in which concurrent object-oriented programs can be written in C++. The concurrent objects in ACT++ are called actors. Specifically, the project is concerned with the development of a communication infrastructure that configures a collection of heterogeneous machines for use in the distributed version of ACT++.

A utility, "ActorControl," was implemented through which the user can specify how ACT++ is to run on each of the nodes by means of a configuration file. The "ActorControl" utility starts a process on each of the nodes specified in the configuration file and establishes TCP socket based connections among all of them. To simplify the communication between the different nodes, a special type of actor called an interface actor is used. Instead of issuing communication requests directly to sockets, a request is directed to an interface actor that is responsible for that socket. A related project is concerned with the problems of creation, destruction and invocation of the methods on the remote machine.

Thus the project consists of two parts. The first part is the implementation of the "ActorControl" utility that establishes socket connections among all the nodes participating in the distributed ACT++. The second part is the implementation of the interface actors that are present at each end of the connection between any pair of machines.

ACKNOWLEDGEMENTS

This work stands testimony to the efforts of many people. I would like to thank all of them who have provided the support and direction necessary to complete my education.

I would like to express my thanks and appreciation to Dr. Dennis Kafura, chairman, for his patience and guidance through the duration of this project. Thanks are extended to Dr. James Arthur and Dr. Scott Midkiff for serving on my committee and giving suggestions to improve my project report.

I would also like to thank Dr. Verna Schuetz for giving me excellent guidance, especially during my first year at Virginia Tech.

I am deeply indebted to Dr. Sriram Pemmaraju whose friendship and technical assistance has helped me in the completion of my Master's degree. He has always been there whenever I needed his help. With his guidance I have been able to overcome many hurdles that I have faced in the past four years.

I thank Mani and Nivedita for helping me in many ways. Without their help, I would not have been able to play at the Commonwealth Games of Virginia, where I won the Gold medal in table-tennis. Mani has also helped me a lot on my project.

Arif and Randy have been great friends. Arif's guidance during the data structures course that I took in summer 1991 helped me in gaining confidence in the field of computer science. Randy offered his help when I needed the most and he found the bug in my program that was eluding me since a long time.

Jay Wang, who sat in the carrel in front of mine for last one year always helped me with a smile. I am grateful to Arjun and Venki for giving me some useful suggestions for my presentation.

I also acknowledge the constant encouragement and company of my friends. I thank

Tejas and my room-mates, Rahul and Sabuji, for putting up with the vagaries of my behavior. Many thanks to Sharathbhai for helping me with Latex. KD and Viraj have been wonderful friends. They have helped me in more than one way.

I would like to thank all my relatives for their confidence in me and their support throughout my educational career. I would especially like to thank Lalitmama, Manishmama, Tanmanben, Ramesh uncle, Jagatbhai, Vikrambhai, Asitbhai, Nipa, Adhir and Anima for the encouragement that they have given to me during the course of my studies at Virginia Tech.

Last and foremost, I am indebted to my parents and sister, Sonali, for their unconditional support and understanding. Their love and support has been a source of inspiration in my work. I could not imagine myself being in this position without their hardwork and perseverance.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

An object-oriented concurrent programming environment seeks to combine the features of the object-oriented approach with the computational power associated with parallelism. Such an environment consists of a collection of independent objects which execute concurrently and communicate with each other via message passing. ACT++ is one such framework in which concurrent, object-oriented programs can be written in C++. ACT++ is based on the actor model of computation which is descriped in Chapter 2.

The project is an attempt to develop a communication infrastructure that configures a collection of heterogeneous machines for use in the distributed version of ACT++. It consists of two parts. The first part deals with the establishment of socket connections between the different nodes participating in distributed ACT++. The second part consists of the implementation of interface actors to simplify the communication between different nodes.

In the remainder of the report, Chapter 2 discusses the important concepts of distributed object-based programming systems. Chapter 2 also presents the actor model and a description of ACT++, which is an implementation of the actor model in C++. PVM and the host-control utility of PCN, which are related works are also decribed briefly. Chapter 3 describes in detail the development of the communication infrastructure. Finally, Chapter 4 discusses the capabilities and limitations of the environment created. Some enhancements are also suggested for future work.

Chapter 2

Background and Related Work

With the widespread use of computers, software applications are becoming more complex and are increasing in size. The computing environments are becoming increasingly distributed and heterogeneous and hence there is a growing demand for new software techniques that would ease the development and maintenance of distributed applications. Object-oriented paradigms have emerged as one of the most promising paradigms for the design, construction and maintenance of such applications. This chapter starts with the discussion of distributed object-based programming systems. The actor model of computation and ACT++, an actor based framework, are discussed next. Finally, two of the existing distributed systems, PVM and PCN, are described briefly.

2.1 Distributed Object-Based Programming Systems

An object-based programming language supports the design and creation of a program as a set of autonomous components, whereas a distributed 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.*

According to Chin and Chanson [CC91], a distributed, object-based programming system has the following features.

Distribution. A distributed, object-based programming system provides a decentralized computing environment by combining a network of independent, possibly heterogeneous machines.

Transparency. The system may hide the underlying details of the distributed environment

from the users. For example, it can provide the feature of local transparency in which the user does not have to worry about the machine boundaries and the physical location of the objects in order to make invocations on them. It should also provide uniform access to all the objects of the system whether they are active in the main memory or inactive in secondary storage.

Data Integrity. A distributed, object-based programming system must ensure that an object is always in a state that is a result of the successful termination of an operation. This means that if an operation fails to complete successfully, the system restores the object's state to its valid state before the operation.

Fault Tolerance. The failure of a machine should not cause the whole system to stall. The loss should be restricted only to that machine. The rest of the system should continue to offer its services and should not be affected by the machine that failed.

Availability. The system must ensure that all the services remain available with high probabilty. In case of any failure, the entire system may be shut down and restarted in order to restore all the services.

Recoverability. If any machine fails, the system must automatically recover the objects that resided on it.

Object Autonomy. The system may permit the owner of an object to specify the clients that have the authority 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. This is not true concurrency unless an object resides in a multiprocessor, since only one request can be processed at a time.

Improved Performance. A well-designed program can typically execute faster than in a conventional system.

An ideal distributed object-based programming system should provide all of the above features. At present, however, no single system provides all of these features. Many of the features provided by a particular system will depend on the intended application domain of the system or the system's design goal. Our system does not provide all of the above features. It provides the features of distribution, transparency, object autonomy, program concurrency and object concurrency. The system is not fault tolerant and hence it does not provide the features of availability and recoverability. It also does not provide data integrity.

2.1.1 Object Structure

The overall design of the distributed, object-based programming system is influenced by the structure of the objects supported by the system. This section defines the three types of objects that can be supported by the system and two ways they can be composed.

Granularity

The relative size, overhead, and amount of processing performed by an object characterizes its *granularity.* The three types of objects supported by distributed, object-based programming systems are as follows.

- Large-Grain Objects: Large-grain objects are characterized by their large size, and relatively few interactions they have with other objects. Large-grain objects reside in their own address spaces.
- Medium-Grain Objects: Medium-grain objects are smaller in size and scope than large-grain objects. They can be created and maintained relatively inexpensively. 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, and relatively large number of interactions they have with other objects.

Composition

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

Passive Object Model: In the *passive object model*, the processes and objects are separate entities. The processes are separate and temporarily bound to the objects they invoke. A process is not restricted to a single object. A single process is used to perform all the operations required to satisfy an invocation. 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. When the process completes this subsequent operation, it is returned to the first object, where it resumes the execution of the original operation. This sequence of events is illustrated in Figure 2.1.

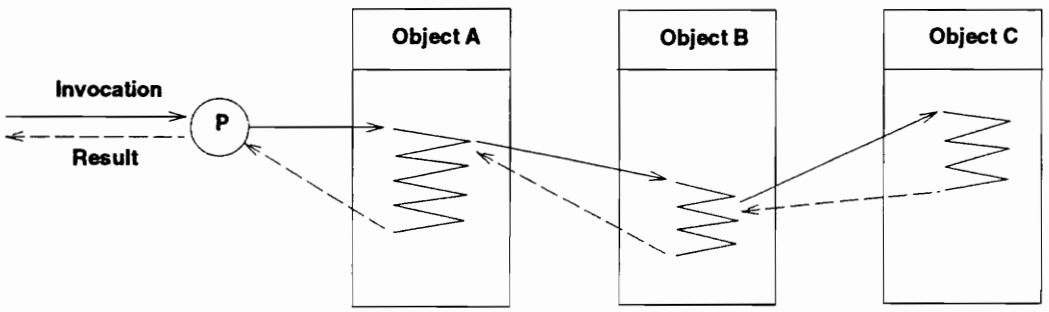


Figure 2.1: Performing an action in the passive object model.

The advantage of using the passive object model is that there is no restriction on the

number of processes that can be bound to an object. The disadvantage of this model is that mapping a process into and out of the address space of multiple objects can be difficult and expensive.

Active Object Model: In an *active object model*, several server processes are created for and assigned to each object to handle its invocation requests. Each process is bound and restricted to the particular object for which it is created. When the object is destroyed, its processes are also destroyed. In this type of model, an operation is not accessed directly by the calling process, as in the case of a traditional procedure call. The client presents an invocation request, together with a list of arguments, to the appropriate object specifying the operation to be invoked. The server object accepts the request and performs the specified operation. While executing an operation, if 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. Finally, when the operation completes, the server returns a result to the client. This approach is illustrated in Figure 2.2.

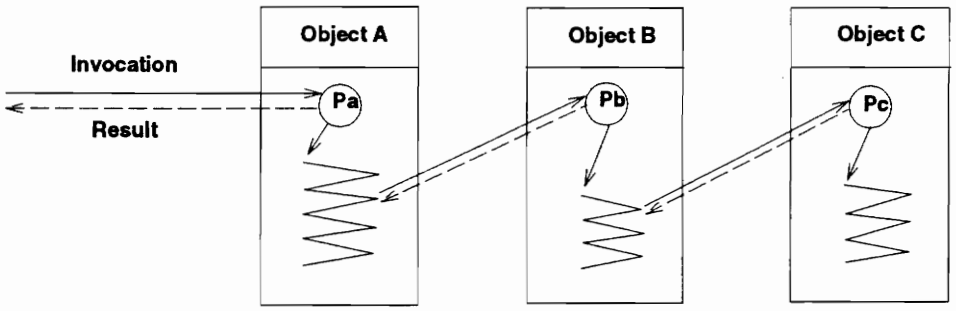


Figure 2.2: Performing an action in the active object model.

2.1.2 Object Interaction Management

A distributed, object-based programming system is responsible for managing the invocations between cooperating objects. When there is an invocation request, the system must locate the specified object, take the appropriate steps to invoke the specified operation and then return the result.

Locating an Object

In a distributed, object-based system, when an invocation is made, the system must determine which object was invoked and where the object currently resides in order to deliver the request to it. The client does not have to be aware of the physical location of the object in order to invoke it. The system must assign an identifier to each object. These identifiers must be unique and should remain constant during their lifetime.There are three approaches for locating the object. They are described below.

Encoding Scheme: In this scheme the location of the object is *encoded* within its object identifier. When an invocation is made, the system simply examines the appropriate field of the specified object identifier in order to determine the location of the object. This scheme is easy to implement and is also efficient. The disadvantage of this scheme is that the objects cannot migrate to another machine as this would require its identifier to change.

Distributed Name Server Scheme: In this scheme, the system creates a group of name server objects that are maintained on a number of machines. There are two variations to 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 passed on to the next. The problem with this scheme is that at least one server must be notified when a new object is created or an object moves from one machine to another. Another drawback of this scheme is that the creation and destruction of an

object may induce significant delay, either to keep the servers up-to-date or to find the server to service the request.

Cache/Broadcast Scheme: In this scheme a small cache is maintained on each machine that records the last known locations of the recently referenced remote objects. When a client makes an invocation request, the cache is examined to determine if it has an entry for the invoked object. If a location is found, the request is sent to that machine. If the object no longer resides on that machine, the request is returned. 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 that receives this request does an internal search for the specified object. If the object is found, a reply message is returned to the machine that made the request and its cache is updated. This scheme is efficient as an object's location may be found in the local cache. It also allows the objects to migrate from one machine to another. The drawback of this scheme is that the broadcast requests would disturb all the machines even though only one machine is directly involved with each location request.

System-Level Invocation Handling

When a client makes an invocation request, the distributed, object-based programming system should perform the necessary steps to deliver the request to the specified server object and return the result back to the client. The two schemes that are used are *message passing* and *direct invocation.*

Message Passing Scheme: A system that provides the active object model supports the *message passing scheme.* When a client makes an invocation request on an object, the parameters of the invocation are packed into a request message and sent to the server process associated with the invoked object. A server process in the invoked object accepts the message, unpacks the parameters, and performs the specified operation. Finally, when

the operation is completed, the result is packed into a reply message, which is sent back to the client. The drawback of this scheme is that it involves overhead of message passing for invocation of the objects residing on the same machine.

Direct Invocation Scheme: A system that provides the passive object model supports the *direct invocation scheme.* When a process invokes a server object that resides on the same machine, the method is directly invoked as in a centralized system. The method invocation is then similar to a function call. When a process invokes a server object that resides on a different machine, a message containing the parameters of the invocation is sent to the machine on which the server object resides. The machine receives this message and 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 sent back to the machine on which the original process resides. The worker process is then killed. This scheme incurs less overhead for local invocations. For remote invocations this scheme has the added overhead of creating and destroying worker processes.

Detecting Invocation Failures

An invocation failure may be classified as being either an *existing fault* or a *transient 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 fault are easy to detect and handle.

A transient fault is a failure that occurs while an invocation is being performed. These faults are much more difficult to detect and handle because there are many different ways an invocation can fail. The system should provide mechanisms for both client and server objects of the system to detect and recover from transient failures.

2.2 The Actor Model

The actor model of computation was developed by Hewitt et al. [HBS73] and extended by Agha [Agh86].

In the actor model the primary agents responsible for all activity are active objects called *actors*. Each actor has its own thread of execution. All objects in the system are actors and there can be several actors executing at the same time. Actors communicate via asynchronous messages called *request messages*. Each actor has a *mail address* and a script which is the representation of its *behavior*. The *mail address* is a pointer to a *mail queue* which buffers request messages. Abstractly, we can picture an actor with a mail queue on which all the request messages are placed and an actor machine which points to a particular message in the mail queue. This is represented pictorially in Figure 2.3.

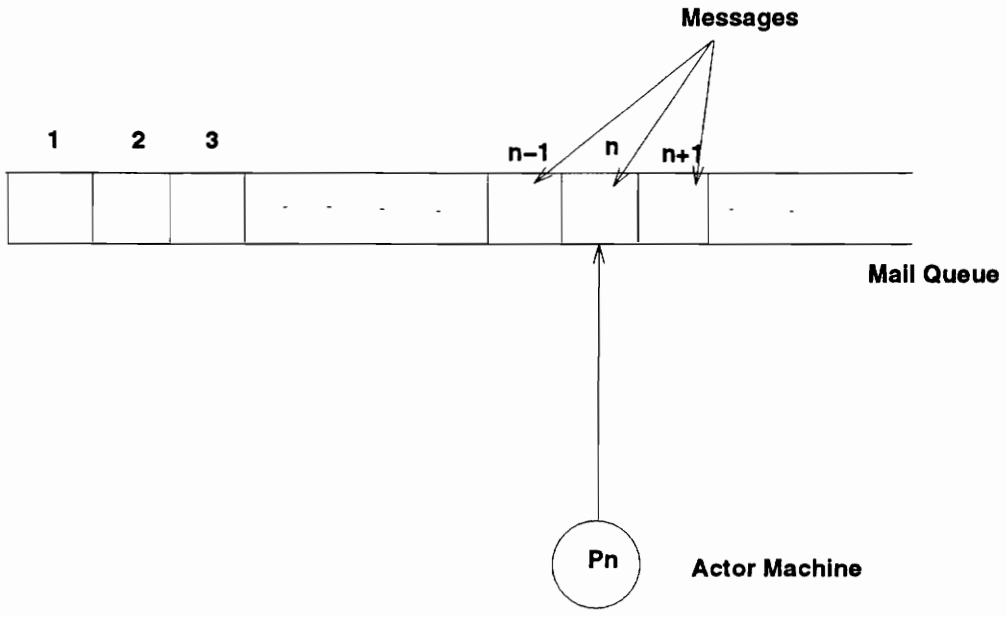


Figure 2.3: Representation of an actor.

Request messages contain the requests for the execution of program segments from

the behavior of an actor. Each behavior is responsible for processing exactly one request message. Request messages are processed by an actor in the order of their arrival, but the arrival order is non-deterministic. The processing of a request message by an actor, as shown in Figure 2.4 could lead to the following actions:

- send request messages to itself or other actors (send),
- create more actors (new), or
- specify the replacement behavior that would process the next message (become).

There is another type of message that is exchanged between the actors. It is called a *reply message*. When an actor sends a request message to another actor, the requested actor sends a reply to the requestor with the result. If the requestor actor needs the reply message, it will not process any of the pending messages in its mail queue. It will resume processing after receiving the reply message.

An actor can communicate with a limited number of other actors for which it knows the mail address by sending messages. Those actors are said to be *acquaintances* of this actor.

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.

As a behavior can process only one message, a behavior must designate, using the become operation, a *replacement behavior*, which will process the next message. The specification of the replacement behavior can be carried out with all other actions in the actor. Also, the replacement behavior is completely independent of the current behavior of the actor. Hence the replacement behavior can immediately start processing an available request message.

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

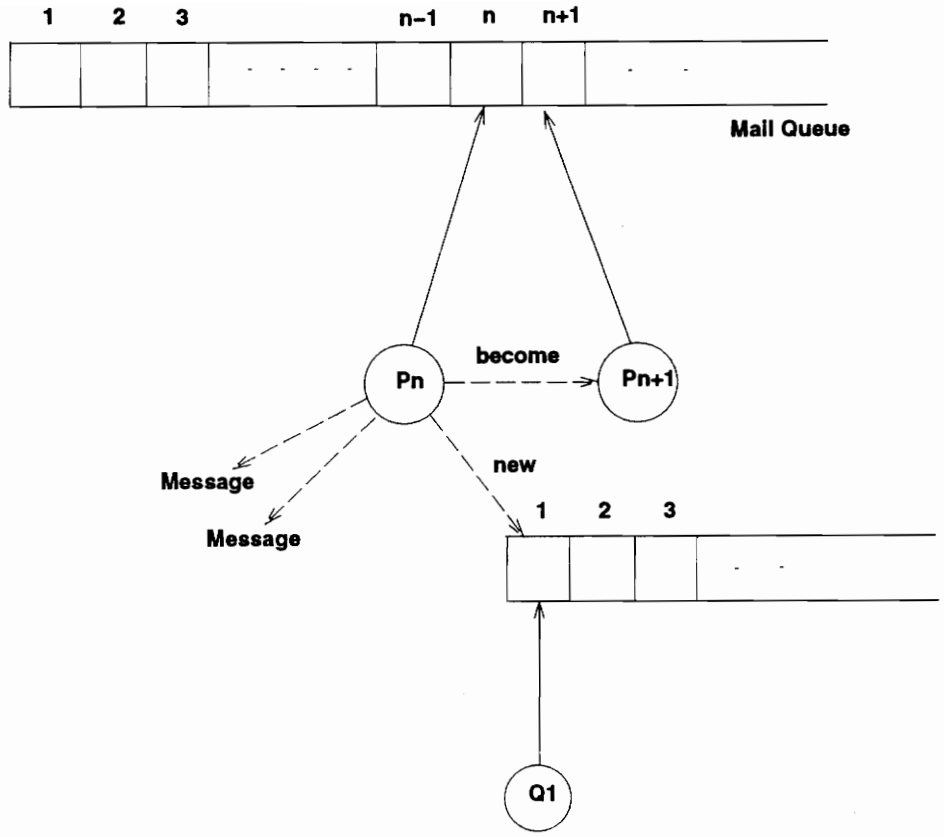


Figure 2.4: Actor operations.

base language or the requirements of the intended application domain. ACT++, which is an actor-based language, is described below.

2.3 ACT++: An Implementation of the Actor Model

ACT++ is an implementation of the actor model in C++. The goal in designing ACT++ was to build a concurrent object-oriented programming environment, an environment that uses the language features of C++ and the program structuring techniques of the actor model. ACT++ does not incorporate all features of the actor model. Instruction-level fine-grain concurrency is not available. 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 programming 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 avaibility of a request message. Request messages are buffered in the message queue of the actor and are processed in a FIFO 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.

2.4 A Distributed Actor System

An actor system is a collection of actors that interact with each other. The graph in Figure 2.5 represents an actor system. Actors are represented by the nodes in the graph. An actor i can send a message to actor j iff there is an arc from node i to node j. All the arcs leaving a particular actor are called the acquaintances of that actor.

A distributed actor system can be implemented in the following three steps. First, an aquaintance is typically a pointer to an actor in a shared memory environment. In the case

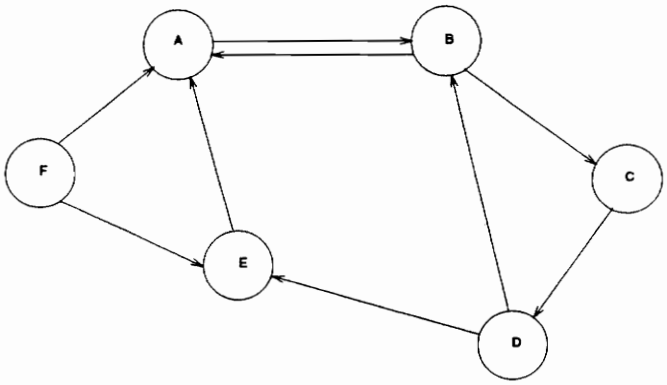


Figure 2.5: An actor system.

of a distributed actor system, acquaintances cannot be implemented with pointers. An actor identifier scheme must be chosen and support is then needed to generate and interpret actor references. Second, in an heterogeneous environment, a mechanism to encode the messages in an architecture independent format is necessary as different machines may implement data types differently. The sender encodes the message and then the encoded message is sent through the network to the receiver which decodes the message. Third, support for dynamic loading and linking of an actor implementation is also necessary. Local invocations and the actor creations are still processed in the same way as for the centralized system.

2.5 The PVM System

PVM stands for *Parallel Virtual Machine.* It is a software system that enables a collection of heterogeneous computers to be used as a single concurrent computational resource [Geist, et al.]. Thus the collection of heterogeneous computers appears as a single large distributed-memory computer. PVM consists of two parts. A daemon process, *pvmd*, that an unprivileged user can install on a machine. When a user wants to run a PVM application, he or she executes pvmd on one of the computers which in turn starts up pvmd on each of the computers making up the user-defined virtual machine. The PVM application can then be started from a UNIX prompt on any of these computers.

The second part of the system is a user library, *libpvm.a*. This library contains routines for communicating between processes, initiating processes, and modifying the virtual machine. Application programs must be linked with this library to use PVM.

The features of PVM are as follows.

- **Machine Reconfiguration**: PVM supplies routines to add and delete hosts from the virtual machine at any time. The master host (the initial pvmd) cannot be deleted from the configuration for the duration of the virtual machine. The master pvmd manages the host table, which needs to be synchronized across all the pvmds. When a new host is added to the virtual machine, a request is sent to the master pvmd. The master pvmd attempts to start and configure pvmds on the new hosts and then broadcasts the new host table to all the other pvmds.

- **Fault detection**: If a host fails, PVM will detect this and convert it into a detectable event, such as a message. This prevents the application from becoming deadlocked. The status of the hosts can be requested by the application and, if required, a replacement host can be added by the application. An exception is the master pvmd. There is no way to recover from the loss of the master pvmd. So, if pvmd loses contact with the master, it exits. Failure of a host is detected when trying to send it a message. The failure to receive an acknowledgement eventually causes the host to be marked dead and the master pvmd edits the failed host out of the host table. Any further requests for the failed host will return with a failure status.

- **Signaling**: PVM provides two methods for signaling other PVM processes. One method sends a UNIX signal to another process. The second method notifies a set of processes about an event by sending them a message. The notification events may be the exiting of a process, the failure of a host, the addition of a host, etc.

- **Communication**: PVM provides routines for packing and sending messages between processes. Any PVM process can send a message to any other PVM process. There is

no limit to the size or the number of messages. PVM also supports multicast to a set of processes and broadcast to a user-defined group of processes. It also provides routines to find out information about the virtual machine configuration and the active PVM processes.

2.6 The PCN System and the Host-Control Utility

PCN stands for Program Composition Notation [FT93]. It is a system for developing and executing parallel programs. Host-control is a utility that is used for the networked version of PCN [OL91]. It configures a user-defined group of machines.

The host-control utility is a Perl script and hence it is necessary to install Perl on each of the machines on which the user wishes to run PCN.

The host-control utility works by starting a daemon program called node-control on each of the user-defined machines that take part in the networked PCN.

When the host-control utility is started, it is ready to accept commands from the user. The user can start the node-control daemons on other machines by using the start-node command. The host-control utility uses rsh to start the daemons on remote machines. The utility also provides a command for terminating the node daemons. One can also find out the status of all the PCN processes running on different nodes by using the "status" command. This will display the process ids of the PCN processes running on the remote machine.

Chapter 3

Building the Communication Infrastructure

This chapter consists of two parts. Section 3.1 describes the "ActorControl" utility, which is a communication infrastructure that configures a collection of machines for use in the distributed version of ACT++. Section 3.2 describes a special type of an actor called an interface actor, which is used to simplify the communication between the different nodes.

3.1 The "ActorControl" Utility

This section focuses on the "ActorControl" utility, which establishes socket connections among all the nodes participating in a distributed ACT++ system. It allows the user to specify how ACT++ is to be run on each of the nodes by means of a configuration file. The "ActorControl" program will read the configuration file and start a background(daemon) process on all the specified nodes. Currently, this utility does not provide a way to start processes on any node dynamically, but this capability can be added in the future. In the event that some of the spawned process has not started successfully, the utility aborts the system by killing all the processes. The utility can also be used explicitly by the user to abnormally terminate the system.

3.2 Use of "ActorControl"

3.2.1 Configuration file

The user can specify the nodes on which the ACT++ system is to be started, along with other details, in the configuration file. The format of the configuration file is as follows. The first field is the node name on which the process is to be started, the second field is the

willow.cs.vt.edu	node1	gandhi	5655	/u1/gandhi/	/u1/gandhi
actor.cs.vt.edu	node2	gandhi	5656	/home/gandhi/	/home/gandhi
csgrad.cs.vt.edu	node3	gandhi	5657	/u2/gandhi/	/u2/gandhi

Figure 3.1: A sample configuration file.

logical name of the node, the third field is the login name of the user on that node, the fourth field is the port number on which the process would wait to accept any connections from other nodes in the configuration file, the fifth field is the directory in which the program file exists and the sixth field is the dynamic search path. A sample configuration file is shown in Figure 3.1.

The configuration file can be edited by starting the "ActorControl" utility with the "-u" option. The user can also specify his or her own configuration file while starting the "ActorContol" program by using the option "-f". Both of these options along with the other options are discussed in Section 3.2.5.

3.2.2 Starting Daemon Processes

The "ActorControl" utility will read the configuration file and for each node specified in the configuration file it will start a process on that node using "rsh". Note that for "rsh" to return after each call, the processes have to be daemonized i.e the processes should run in the background. The technique to daemonize a process consists of having the process call *fork* to create a new process, and then arranging the parent to exit. For "rsh" to be successful the necessary condition is that the *.rhosts* file in the home directory of the user on the remote machine should contain an entry of the node on which the "ActorControl" utility is running and the userid. If "rsh" is succesful then a TCP socket connection is established between the "ActorControl" utility and the daemon process started on the remote node. The "ActorControl" utility sends the configuration file to the remote node and then receives

the process id of the daemon process on the remote node via the socket connection. The process ids are recorded in a file *piddata*. This is illustrated in Figure 3.2.

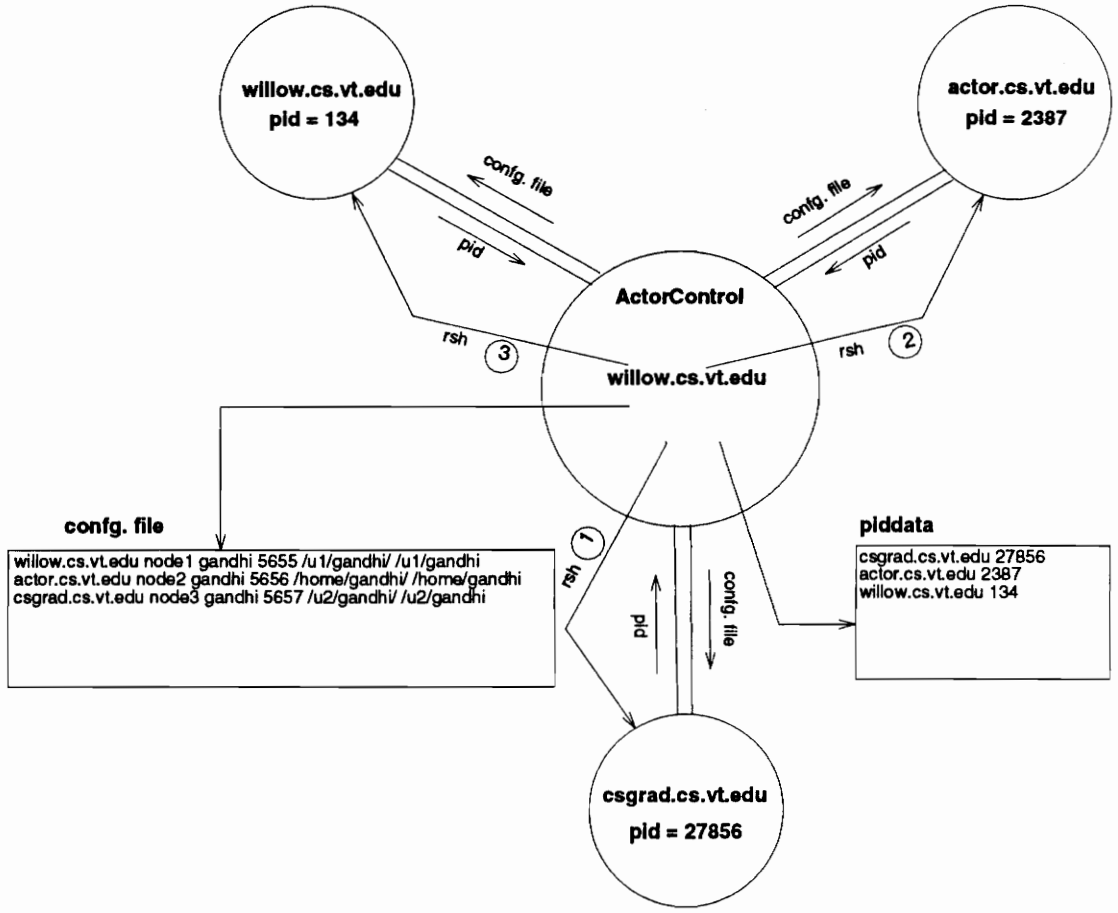


Figure 3.2: "ActorControl" utility starting the daemon processes.

With reference to Figure 3.2, "ActorControl" is started on `willow.cs.vt.edu`. It reads the configuration file and starts the daemon process on `csgrad.cs.vt.edu`, then on `actor.cs.vt.edu` and finally on `willow.cs.vt.edu`. The order in which the daemon processes are started is important. After sending the configuration file and receiving the process ids of the daemon processes, a global error variable is checked. This variable will be set if any of the "rsh" commands failed to complete. In this case, "ActorControl" will

then invoke a program which will terminate all the daemon processes. This is discussed in the next section.

3.2.3 Killing Daemon Processes

The "ActorControl" utility also provides a feature for terminating the daemon processes on all the nodes if some error occurs. As mentioned in the previous section, when some error occurs during the startup phase, a program to terminate all the daemon processes, "killer", is invoked and the file in which the process ids are recorded is passed to it as an argument. The "killer" program reads the file and kills the daemon process running on all the nodes on which "rsh" was successful. The "killer" program can also be invoked from the command line of the machine on which "ActorControl" utility was started and the file which contains the process ids of all the processes running on different nodes is passed as an argument. The "killer" program is usually invoked from the command line when the user decides to end the ACT++ program. An example of the command line invocation is as follows:

```
start-node % killer piddata
```

In the above example "killer" will read the file *piddata* and kill all the processes running on the different nodes whose process ids are recorded in *piddata.*

Figure 3.3 shows that the "ActorControl" utility failed to start the background process on `csgrad.cs.vt.edu` using "rsh", as the *.rhosts* file on `csgrad.cs.vt.edu` did not permit the user on `willow.cs.vt.edu` to start a process using "rsh". This failure is noted by the "ActorControl" utility and, after obtaining the process ids of the daemon processes on the other nodes, the "killer" program is invoked and the file *piddata* is passed as an argument. The "killer" program reads the file *piddata* and executes the "kill" command on each of the nodes specified in *piddata* using "rsh" to kill the daemon process running on that node.

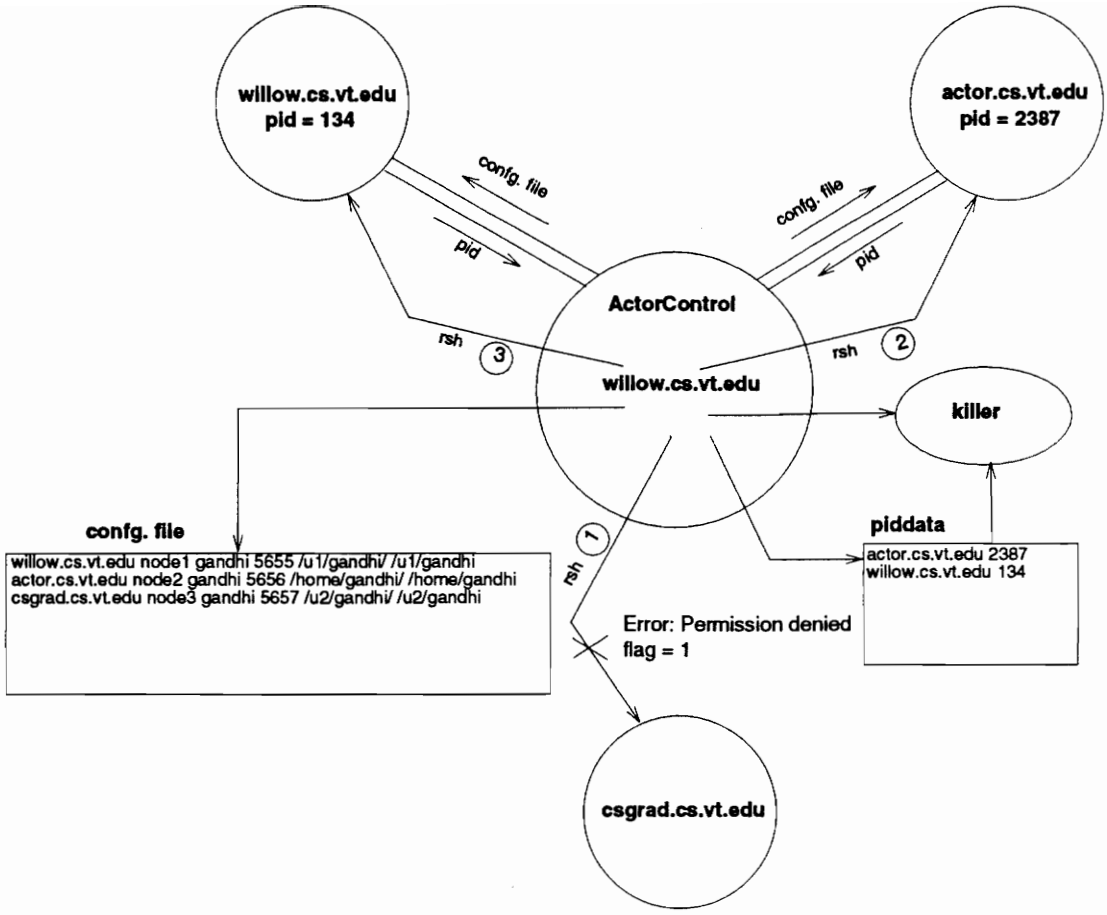


Figure 3.3: Killing the daemon processes.

3.2.4 Forming A Complete Graph

The daemon processes running on all the nodes read the configuration file sent by the "ActorControl" program. Each process attempts to establish a socket connection with all other processes named in the configuration file. The algorithm for establishing the connections is as follows. Assume that there are N nodes to be connected. Each node is identified by its position in the configuration file. For the sample configuration file shown before, N = 3, `willow.cs.vt.edu` is node 1, `actor.cs.vt.edu` is node 2 and `csgrad.cs.vt.edu` is node 3. To establish its connection, node i will (1) accept connections from the first *(i - 1)*

nodes and (2) initiate connections for the remaining *(N - i)* nodes. In the example configuration, `willow.cs.vt.edu` will not wait for any connection, but will initiate a connection with `actor.cs.vt.edu` and `csgrad.cs.vt.edu`. After `actor.cs.vt.edu` accepts the connection request from `willow.cs.vt.edu`, it establishes a connection with `willow.cs.vt.edu` and it also sends out a connection request to `csgrad.cs.vt.edu`. Finally, `csgrad.cs.vt.edu` will accept the requests from `willow.cs.vt.edu` and `actor.cs.vt.edu`. Thus, all the nodes are connected to each other via socket connection. A complete graph is formed in which the vertices are the daemon processes and the edges are the socket connections.

When the connections are established between all the nodes the state of the system is as shown in Figure 3.4.

3.2.5 Other Features

"ActorControl" can be started with several options. They are as follows.

-f *filename*: When started with this option, "ActorControl" will use the *filename* as the configuration file instead of the default configuration file *config.stat.*

-i [hostname]: When started with this option, "ActorControl" will provide the information of the hostname from the configuration file. If the user does not give the hostname the whole configuration file is shown.

-a *filename* hostname username port [pathname]: When started with this option, the host with the relevant details of username, port number and path which is optional is added to the configuration file specified by the *filename.*

-u [*filename*]: When this option is used, "ActorControl" will open the configuration file specified by *filename* for editing. If the *filename* argument is absent then the default configuration file, *config.stat*, is opened for editing. The editor used to edit the file is "vi."

For example, the command "ActorControl -fu config.new" will result in the following action. "ActorControl" will first open the configuration file *config.new* for editing and then after making any changes, if needed, the file is saved and then the daemon processes are

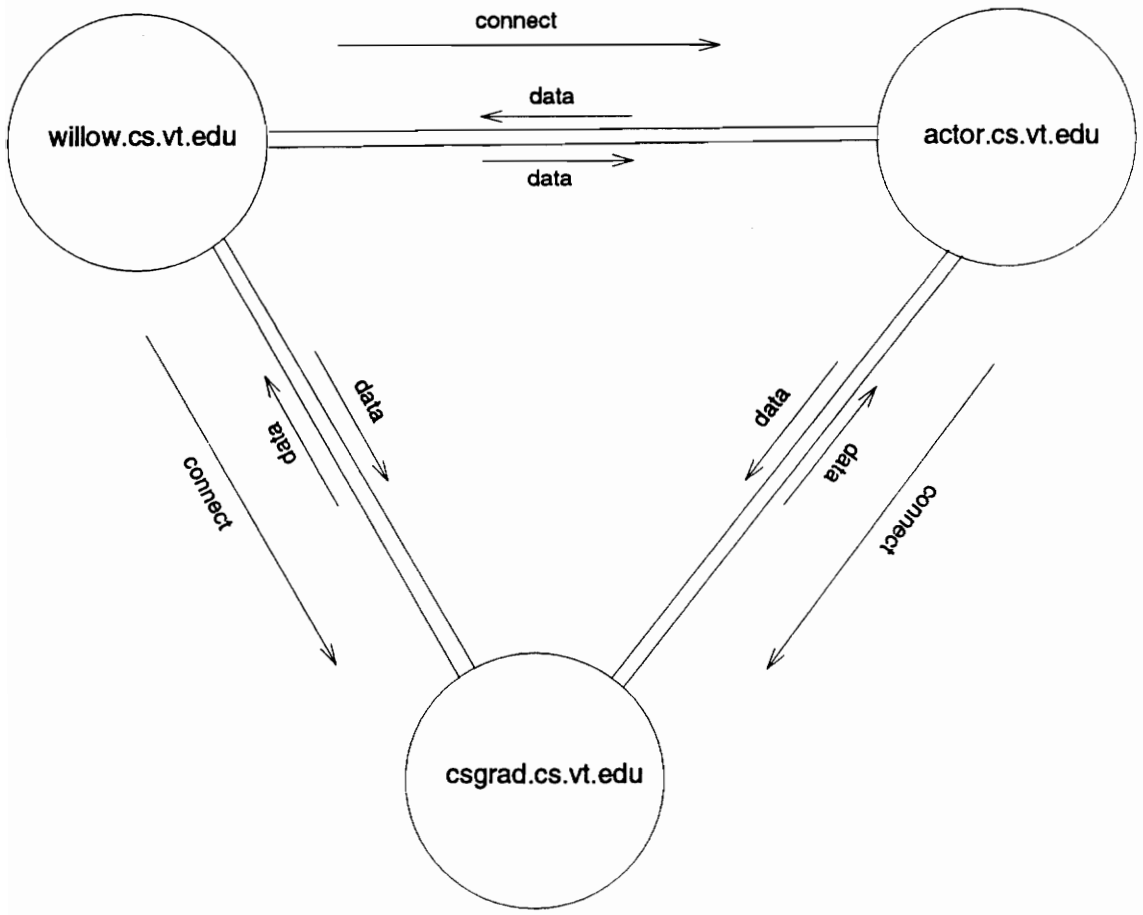


Figure 3.4: Daemon processes forming a complete graph via socket connections.

spawned on the different nodes specified in *config.new*. If *config.new* argument is absent then the default configuration file *config.stat* is opened for editing and the daemon processes are spawned on the nodes specified in *config.stat*.

3.3 Interface Actors

Input/Output (I/O) 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 a request to an actor that is responsible for that socket. Actors which handle I/O to

sockets are called *interface actors* and the *IActor* class in ACT++ implements such actors. The definition of the class follows.

```
class IActor : public Actor {
public:
        IActor(client_beh *int_beh,char *mc_name,int port_no,char *cdir);
        IActor(server_beh *int_beh,int port_no,int num_listen,char *cdir);
        IActor(client_beh *int_beh,int sfd,char *cdir);
        IActor(server_beh *int_beh, int sfd, char *cdir);
}
```

An *interface actor* that is constructed using client_beh will initiate connection while the one that is constructed using server_beh will accept connections. For any given pair of machines participating in distributed ACT++, two *interface actors* would be created, one with client_beh on one of the machines and the other with server_beh on the other machine. Both client_beh and server_beh are inherited from class cli_serv_beh which is a subclass of Behavior. The class definitions of cli_serv_beh, client_beh and server_beh are as follows.

```
class cli_serv_beh : public Behavior {
private:
        char *cdir;
protected:
        int sockfd;
public:
        void put_cdir (char *str);
        void get_mesg (char *buff);
        void read_data (void);
        void write_data (void);
        char *get_cdir (void);
}
```

In class cli_serv_beh, *cdir* is a private member that holds the path name in which the program file exists. *sockfd* represents the socket descriptor and is a protected variable used by subclasses client_beh and server_beh. The member functions read_data and write_data read from and write to the socket, respectively.

```
class client_beh : public cli_serv_beh {
public:
        client_beh (void);
        void cli_open (char *host_name, int port_num, char *cdir);
        void cli_open (int sfd, char *cdir);
}
```

The method cli_open takes a character string which represents the host name to which it is to connect. The port number is represented by port_num. Via operator overloading, cli_open may also take a socket descriptor and a character string as arguments. This method is called from within the *interface actor* constructor when an *interface actor* is created for reading from or writing to a socket. In all the methods, the argument cdir represents the path name of the directory in which the program file exists.

```
class server_beh : public cli_serv_beh {
public:
        server_beh ();
        void serv_open (int port_num, int num_listen, char *cdir);
        void serv_open (int sfd, char *cdir);
        void accept_conn (int node_num);
}
```

There are two serv_open methods. One of them takes a port number on which it accepts connections and num_listen represents the maximum number of connection requests that can be queued by the system while it waits for the server to execute the accept call. The

second method takes in a socket descriptor as an argument. This method is called from within the *interface actor* constructor when an *interface actor* is created for reading from or writing to a socket. The method accept_conn waits for connection requests from other nodes. The argument node_num for accept_conn represents the sequence number of the node in the configuration file, so that the process running on that node knows the number of connections that it has to accept. In all of the methods, the argument cdir represents the path name of the directory in which the program file exists.

The algorithm used by the "ActorControl" utility uses *interface actors* to form connections between all the nodes. The daemon process creates a server behavior using the default constructor and then passes it as one of the arguments to the *interface actor* whose definition is as follows.

```
IActor::IActor (server_beh *int_beh, int port_no, int num_listen,
                                char *cdir) : Actor ((Behavior *)int_beh)
{
        int_beh->serv_open (port_no, num_listen, cdir);
}
```

Within the constructor, method serv_open of the the server_beh is called. The definition of method serv_open is given below.

```
void server_beh::serv_open (int port_num, int num_listen, char *cdir)
{
        int optval = 1;
        struct sockaddr_in serv_addr;

        put_cdir (cdir);
        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                errexit ("server: can't open stream socket");
        }
```

```
        if (setsockopt (sockfd, SOL_SOCKET, SO_REUSEADDR, (char *)&optval,
                                                        sizeof (int)) < 0) {
                errexit ("can't create a socket: %s\n", sys_errlist[errno]);
        }
        bzero ((char *) &serv_addr, sizeof (serv_addr));
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = htonl (INADDR_ANY);
        serv_addr.sin_port = htons (port_num);
        if (bind(sockfd,(struct sockaddr *)&serv_addr,sizeof(serv_addr)) < 0) {
                errexit ("can't bind local address: %s\n", sys_errlist[errno]);
        }
        listen (sockfd, 5);
}
```

In method server_open, a socket is created and is bound to its local address.

Then the daemon process creates a message and sends it to the interface actor requesting the interface actor to execute the accept_conn method from its behavior. The definition of the accept_conn method is as follows.

```
void server_beh::accept_conn (int nodenum)
{
     int newsockfd;
     int clilen;
     int i;
     struct sockaddr_in cli_addr;
     serv_temp<server_beh> int_beh;

     for (i = 0; i < (nodenum - 1); i++) {
        clilen = sizeof (cli_addr);
```

```
        newsockfd = accept (sockfd, (struct sockaddr *)&cli_addr, &clilen);
        if (newsockfd < 0) {
            errexit ("accept failed: %s\n", sys_errlist[errno]);
        }
        else {
            server_beh *read_beh = new server_beh ();
            server_beh *write_beh = new server_beh ();
            IActor *iact = new IActor (read_beh, newsockfd, get_cdir());
            IActor *iact1 = new IActor (write_beh, newsockfd, get_cdir());
            Message *read_mess = int_beh.read_data ();
            Message *write_mess = int_beh.write_data ();
            read_mess->send (iact);
            write_mess->send (iact1);
        }
    }
}
```

The argument passed to this method is the sequence number of the node in the configuration file. This node waits for a connection from all the nodes listed before it in the configuration file. After each connection is accepted two interface actors are created, one for reading the data, and another for writing the data to the other end of the connection.

Next, the daemon process creates an interface actor for every node listed in the configuration file after it. The definition of the constructor for the *interface actor* is as follows.

```
IActor::IActor (client_beh *int_beh, char *mc_name, int port_no,
                                char *cdir) : Actor ((Behavior *)int_beh)
{
        int_beh->cli_open (mc_name, port_no, cdir);
}
```

The above constructor calls the cli_open method in the client_beh. The definition of cli_open is given below.

```
void client_beh::cli_open (char *host_name, int port_num, char *cdir)
{
    struct sockaddr_in serv_addr;
    struct hostent *host_ptr;
    cli_serv_temp<client_beh> int_beh;

    put_cdir (cdir);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons (port_num);
    if ((host_ptr = gethostbyname (host_name)) == NULL) {
            errsys ("%s is an unknown host\n", host_name);
    }
    bcopy ((char *) host_ptr->h_addr,(char *) &serv_addr.sin_addr,
                                                    host_ptr->h_length);
    if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
            errexit("can't create socket: %s\n",sys_errlist[errno]);
    }
    put_cdir (cdir);
    if (connect(sockfd,(struct sockaddr *)&serv_addr,sizeof (serv_addr)) < 0) {
            errexit ("connect failed: %s\n", sys_errlist [errno]);
    }
    client_beh *read_beh = new client_beh ();
    client_beh *write_beh = new client_beh ();
    IActor *iact = new IActor (read_beh, sockfd, get_cdir());
    IActor *iact1 = new IActor (write_beh, sockfd, get_cdir());
    Message *read_mess = int_beh.read_data ();
```

```
    Message *write_mess = int_beh.write_data ();
    read_mess->send (iact);
    write_mess->send (iact1);
}
```

In the above method, a socket is created and a connection request is sent to the node given by hostname. Once a connection is established, two interface actors are created, one for reading the data and another for writing the data to the other end.

The state of a complete actor system is shown in Figure 3.5. As shown in the figure, two interface actors are created at each end of the connection, one for reading the data and the other for writing the data to the node at the other end of the connection.

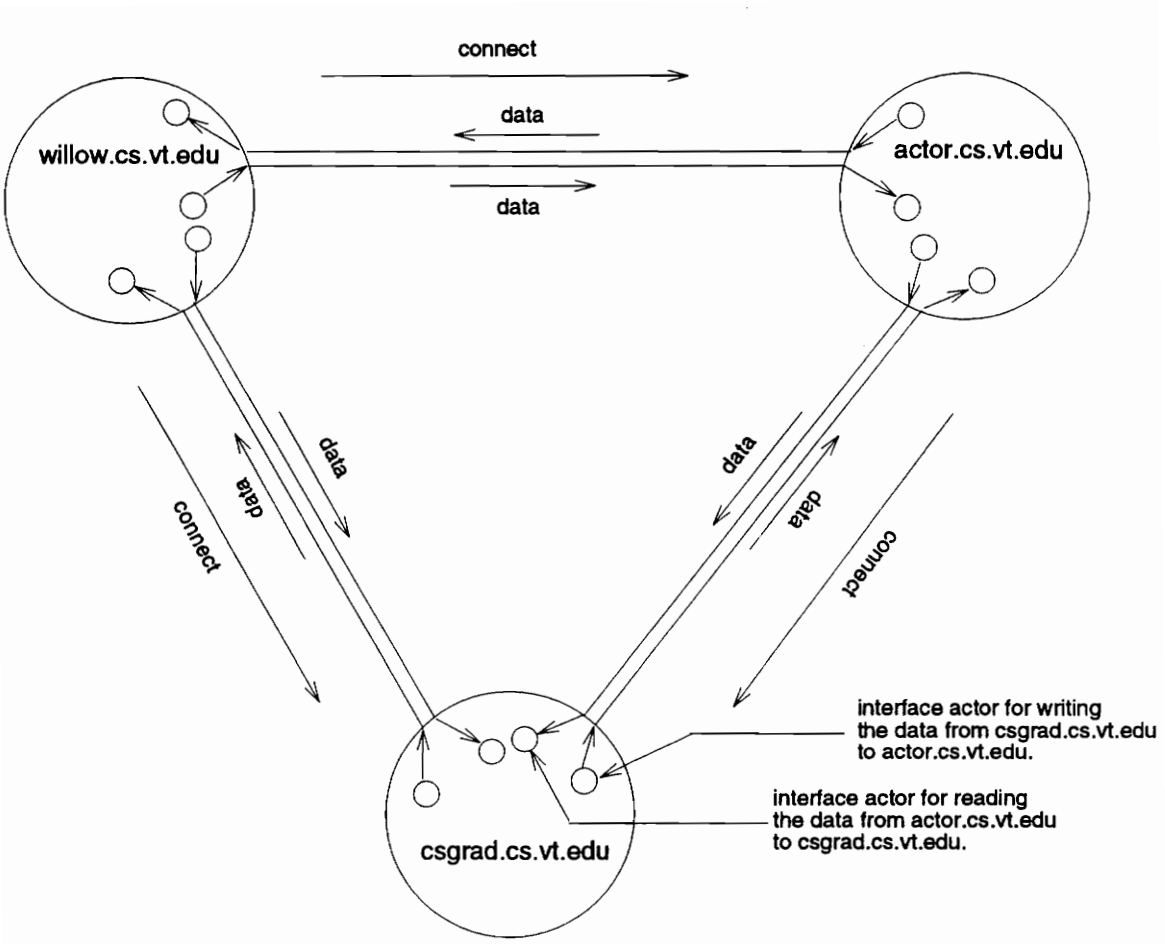


Figure 3.5: State of the system with interface actors at each end of the connection.

Chapter 4

Conclusions and Future Work

The system described in this report:

- establishes socket connections between all the nodes specified in the configuration file using the "ActorControl" utility, and
- sets up interface actors at each end of the connection to simplify the communication between the nodes.

These steps are sufficient to establish a communication structure for a distributed ACT++ computation. The "ActorControl" utility starts the process on each of the nodes specified in the configuration file. The configuration file is sent to all the nodes and the process ids of all the processes are recorded. Then all the nodes specified in the configuration file are connected to each other via TCP socket connections.

The current system has several limitations. Three limitations and suggested remedies, are now discussed.

First, the utility allows only the nodes specified in the configuration file to take part in distributed ACT++. It does not provide a way to include any node dynamically. The current system can be extended to permit dynamic addition of new nodes. This extension is now briefly discussed.

After establishing connections among themselves, an interface actor on each node will wait on the accept system call for additional connections. To add a new node to the configuration, the node should be added to the configuration file and a process should be started for the new node using "rsh". Then the configuration file should be sent to the new node and the process id of the process on the new node should be recorded. The daemon

process on the new node can get the nodes on which the other processes are running from the configuration file. All the processes are waiting on the accept call using a predetermined port number. Hence connection can be established between the new node and the other nodes using the connect system call. After establishing connections with the other nodes the process on the new node will wait on the accept system call waiting for any connections from the nodes added to the configuration after itself.

Second, the nodes are connected to each other via TCP socket connections. This provides a *connection-oriented service* which has the advantage of reliable message transfer, but has the overhead of forming *N-1* connections, where *N* is the number of nodes participating in distributed ACT++. Instead of providing a *connection-oriented* service, a *connectionless* service, also called a *datagram* service, could be provided by using UDP. In a *connectionless* service the client does not establish a connection with the server. The client uses the `sendto` system call which requires the address of the server as a parameter. Similarly, the server does not have to accept a connection from a client. The server just issues a `recvfrom` system call that waits until data arrives from some client. The `recvfrom` returns the network address of the client process, along with the datagram, so the server can send its response to the correct process. The "ActorControl" utility would not require to make any connections if connectionless service were used. But this service is not reliable. If a datagram disappears neither the client nor the server are aware of it. Some kind of timeout and retransmission strategy would needed in the interface actors to make the message transfer reliable.

Third, the "ActorControl" utility requires the user to specify the port numbers on which the nodes should accept connections from other nodes in the configuration file. The drawback of this way of specifying the port numbers is that the port number might be used by some other user. In that case, the "ActorControl" utility will fail. The user would have to change the port number in the configuration file for the "ActorControl" utility to work. The utility could be made more robust by obtaining the port number from the system as the system would give the port number that is not used by anyone else on that node.

Appendix A

The "ActorControl" utility

Name:

ActorControl - forms connections between all the nodes participating in distributed ACT++.

Syntax:

ActorControl [options]

Description:

The "ActorControl" utility configures a collection of machines for use in the distributed version of ACT++. The utility establishes socket connections between all the nodes participating in distributed ACT++. It allows the user to specify how ACT++ is to run on each of the nodes by means of the configuration file. Each line of the configuration file contains the relevant details for each node. There are six fields on each line. The first field is the node name on which the process is to be started, the second field is the logical name of the node, the third field is the login name of the user on that node, the fourth field is the port number on which the process would wait to accept any connections from other nodes in the configuration file, the fifth field is the directory in which the program file exists and the sixth field is the dynamic search path.

Options:

"ActorControl" can be started with several options. They are as follows.

-f *filename*: When started with this option, "ActorControl" will use the *filename* as the configuration file instead of the default configuration file *config.stat.*

-i [hostname]: When started with this option, "ActorControl" will provide the information of the hostname from the configuration file. If the user does not give the hostname the whole configuration file is shown.

-a *filename* hostname username port [pathname]: When started with this opti on, the host with the relevant details of username, port number and path which is option al is added to the configuration file specified by the *filename.*

-u [*filename*]: When this option is used, "ActorControl" will open the configuration file specified by *filename* for editing. If the *filename* argument is absent then the default configuration file, *config.stat,* is opened for editing. The editor used to edit the file is "vi."

```
example % ActorControl -fu config.new
```

"ActorControl" will first open the configuration file *config.new* for editing and then after making any changes, if needed, the file is saved and then the daemon processes are spawned on the different nodes specified in *config.new.* If *config.new* argument is absent then the default configuration file *config.stat* is opened for editing and the daemon processes are spawned on the nodes specified in *config.stat.*

REFERENCES

[Agh86] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, MA, 1986.

[C93] Gilles Carlo. Dynamic Loading and Class Management in a Distributed Actor System, M.S., Project Report, Department of Computer Science, Virginia Polytechnic Institute and State University, July 1993.

[CC91] Roger S. Chin and Samuel T. Chanson. Distributed Object-Based Programming Systems. *ACM Computing Surveys*, 23(1):91-124, March 1991.

[CO93] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP, Volume III.* Prentice-Hall, Englewood Cliffs, NJ, 1993.

[FT93] Ian Foster and Steven Tuecke. Parallel Programming with PCN, Version 2.0, Argonne National Laboratory, University of Chicago.

[Geist, et al.] Al Geist et. al. PVM 3 User's Guide and Reference manual, Oak Ridge National Laboratory, Oak Ridge, TN, 1993.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A univeral modular actor formalism for artificial intelligence. In *Proceedings of the 3rd IJCAI*, pages 235-245, 1973.

[KER90] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1990.

[MUK92] Manibrata Mukherji. The Implementation of ACT++ On A Shared Memory Multiprocessor, M.S., Project Report, Department of Computer Science, Virginia Polytechnic Institute and State University, February 1992.

[OL91] Robert Olson. Using host-control, Argonne National Laboratory, University of Chicago.

[R93] Stephen A. Rago. *UNIX System V Network Programming.* Addison-Wesley, Reading, MA, 1993.

[ST92] W.Richard Stevens. *UNIX NETWORK PROGRAMMING,* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[STR91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Menlo Park, CA, second edition, 1991.