1988

Implementation of a module implementor for an activity based distributed system

Stewart W. Mayott

Rochester Institute of Technology
School of Computer Science and Technology

Implementation of a Module Implementor for an Activity Based Distributed System

by

Stewart W. Mayott

February 14, 1988

A thesis submitted to the Faculty of the School of Computer Science and Technology, in partial fulfillment for the degree of Master of Computer Science

Approved by:

Dr. James E. Heliotis

Prof. Guy Johnson

Prof. Margaret M. Reek

Stewart W. Mayott
13 Fitzpatrick Rd.
Grafton, Ma. 01519
April 1, 1988

To whom it may concern:

I have given to the library of Rochester Institute of Technology, Rochester, N.Y. a copy of my thesis, "Implementation of a Moldule Implementor for an Activity-Based Distributed System". I further grant permission for the institute, their representatives, or any requesting persons to make copies of this thesis with no need to inform me or to reimburse me for such actions.

Sincerely,

Stewart W. Mayott

This thesis investigates process environment within a distributed computational network, in particular, an activity-based distributed system.

An activity-based distributed system is an object-oriented programming system. Objects are grouped together into logical arrangements called activities as the programmer desires. All program code is contained within objects, and some objects are implementors of object types. Objects in an activity receive activity information related to other objects in the activity and have virtual communication capability with other objects in the same activity.

To investigate the feasibility of this environment, a fully functional module implementor was created. Further, other components with limited functionality were coded. An assembly line was simulated within the context of these components. Object death and communication were built into the simulation to investigate this functionality and results upon the system.

Stewart Mayott

1. INTRODUCTION

As the utilization of networks and the demand for processor availability increase, I believe the current practices of distributed systems must change. If there is a method or system to facilitate the development and proper execution of multiple tasks distributed at various sites of a system, industries may employ more computer systems for supervisory control. Two important areas of concern are the task hierarchical structure and transparent task communication.

Further, these distributed systems must be capable of handling the failure prevalent on most assembly lines. Better yet, the distributed system must facilitate and offer acceptable tools to build a fault tolerant application.

The proposed system targets these issues such that the application implementor's main concern is not these issues but the goal of the application.

I will discuss the past and current developments of distributed systems in Chapter Two. Chapter Three gives the terminology and precepts of a proposed activity based distributed system. I will outline my design for a Module Implementor in Chapter Four. Chapter Five presents a test case for the the implementor and describes a system which I have built for this testing. Complications and difficulties incurred during the implementation cycle are discusssed in Chapter Six. Chapter Seven contains related topics for future enhancements or future thesis topics and the conclusion. I discuss my results, observations, and conclusions in Chapter Eight.

2. BACKGROUND - DISTRIBUTED SYSTEMS

A current problem with the development of distributed systems is the underlying communication and process management that is necessary. For most widely used operating systems, both need to be coded by the developer of the system. In this chapter, I would like to discuss traditional methods that have been employed for these purposes. Then, I will discuss a newer system whose designers attempted to alleviate these limitations. I conclude this chapter with an overview of a proposed activity-based distributed system.

2.1. Traditional Methods

The traditional approach for the development of distributed systems has been to build a slave/master or server/client relationship. A server is a segment of code that listens at a well-known address for any process which would like to be able to receive service from the listening process. When a requesting process connects to the server, this process, the client, may then proceed with the execution of its task or request the server to perform a task on the behalf of the client process. Performing a connection to a process creates a logical end-to-end communication between the two processes, allowing the data to be sent from one process to the other without the underlying data transfer information being specified for each transfer request. This server, the code to connect to the server, and the request for a particular type of service, need to be written by the developer for any particular application.

Under the UNIX* operating system, the servers must be written, placed on the system and registered with all systems. This enables any process desiring that service to determine its address and connect to that service. Two difficulties with this approach are: the communication must be via virtual circuits, and the service must do its own management of those circuits. Currently, the only protocol implemented in UNIX that will guarantee the order of delivery of the packets is socket stream (datagrams do not guarantee the order of the packets, and sequential packets are not yet implemented) [UNI83]. This means that once a client connects to the server, the client has the server to itself, but the result becomes another problem, that no other processes can receive service. Of course, the developer could develop a system under which a pseudo-server does the listening and is able to start various versions of the actual server. Yet, this would require application management of socket communication and/or processes. In either case, the developer is not truly developing the distributed system but the underlying architecture to handle the system. Finally, there is no method by which a developer may simply define interdependencies between processes on various systems. The developer is again forced to build a system to handle any type of special relationships and/or failure of a process or processes. There may be a parent-child relationship where the two processes seem to be bound to each other for some related purpose. Yet, as most people who have worked with this type of system can tell you, this is not always the case. You may want to end the relationship and perhaps start another with a process that is already started, whose common ancestor is far removed.

---

*UNIX is a trademark of Bell Telephone Laboratories, Inc.

A similar approach is used by the Courier Remote Procedure Call Protocol [XER81]. The Courier socket on every machine is well-known, and only one Courier dispatcher may reside on each machine at a time. Courier takes care of the underlying network calls (virtual circuits) between the machines. The developer need only define the type of values returned and error signals to be sent to the requesting machine. Unfortunately, for the client, the remote procedure calls need to be sequential in execution, not concurrent. Again, there is no method by which a developer may define system-wide relationships between programs and objects on separate machines.

2.2. Developed Systems

The Argus system developed at MIT is one that has been designed to overcome the obstacles associated with distributed system development. It is composed of a language and an operating system. The tasks are split up and distributed to the various physical nodes by language objects called guardians. Guardians can be thought of as virtual nodes since they share no data. Control information and other data are transferred using remote procedure calls which communicate to handlers. The operating system provides built-in atomicity for the handler call, including a two-phase commit protocol and error recovery. Each top-level handler call is called an action and must totally succeed or be aborted.

Thus, for distributed systems which have fixed goals for each program in the system, the architecture and management tools for building a distributed system exist. Argus takes care of the communication difficulties and failure of a physical node or the software failure of a node.

Consider the following characteristics for a data base management system:

1) Computations are to be atomic;
2) Objects are long-lived and dedicated to one task; and
3) Objects involved in the same task reside at the same physical node.

These characteristics can all be satisfied by the Argus system. But, consider an addition of a printer to this management system. The printer is a long-lived object but its involvement in a specific task may be short-term, or the print server may not be at the same physical location. In this case, the print server cannot be declared inside the guardian, because all the objects registered in a guardian must be on the same machine. Further, how can a back-out be performed on a print command? If part of a print command is performed, but must abort, what exceptions must be made within the system? This case may be true for certain other environments and devices. Therefore, the needs of the application developed for that environment are not met.

2.3. An Activity Based System

The development of an activity-based distributed computational system is slightly different than for traditional development approaches. To develop a system, the developer designs activities, which are defined as "dynamic, identifiable collections of state information that are spread among a dynamically changing set of objects in a computational network" [HEL84]. In other words, an activity is composed of a dynamically changing set of objects that cooperate to accomplish a single goal. Each activity can have an Activity Control Module to handle activity commands and

respond to emergency messages. Unlike the guardian system which controls a single node, an activity may involve objects throughout the system.

The system may be developed in any language, but the original thesis was described utilizing the Mesa Language, with added constructs. This language and added constructs allow concurrent processing to occur. A module may be INSTANTIATED to create the execution of another module, but the calling routine will not wait until the completion of the INSTANTIATED module. The underlying system handles the correct completion of the module. Failure of the module completion may be communicated to an Activity Coordinator who may then inform other interested parties or be capable of restarting the module.

The complete system is defined and described in Chapter Three, and an example is described to aid in its understanding.

3. ACTIVITY BASED DISTRIBUTED SYSTEM

In this chapter I will define and explain more fully the terms that have thus far been used very loosely. A scenario and an example are given at the end to aid in the understanding of the system.

3.1. Definitions

The terminologies and constructs of the system are defined as follows:

object An element in the system which maintains an internal state. The state of the object may be divided among the various activities in which it is involved. An abstraction of its function can be made available to the outside world. Examples of objects are process modules and external device controllers.

object manager An object that performs the object-specific interpretation of commands that are related to activity management rather than the application itself. Objects can be self-managing.

module Object that contains code, data, and process state and is capable of communicating to the outside world. Modules are the basic object type of the system (all other object types are user-defined) and are self-managing.

object implementor Module external to an object that can create, destroy, and possibly perform routine aspects of maintaining the object.

module implementor Object that starts, kills and informs the activity coordinator of the death of a module. The implementor also handles the initiation of module communication.

activity A dynamic, identifiable collection of state information that is spread among a dynamically changing set of objects in a computational network.

activity control module Module that oversees an activity by working with the activity's objects in response to activity commands and emergency messages.

activity coordinator Module that maintains the various components and registration lists and routes activity-related messages.

The activity-related communications of the system are hidden from requesting and receiving objects. They are handled by either the object manager or the object implementor or both. It should be noted that an object's manager and implementor may be user-defined "code" that manipulates other objects, as in the case of a file manager or device driver.

3.2. Roles

James Heliotis, in [HEL84], identifies roles that are associated with objects and activities as a dynamically created tag to identify the purpose of the collection of objects within an application. I have seen a need for a module role within these activity "roles". This information is not used in any fashion by the activity system to maintain the system components. This information is specified upon creation by the requesting object. It informs the module as to the purpose of its existence and for recognition between objects within the same activity. It can identify a particular task with the goal of the activity. Contrary to the activity tag or an object identifier, which are dynamically created by the system, the module role may be set by the application implementor. This allows the application implementor a mechanism to efficiently process requests and forward requests to specific modules within its activity.

This functionality and information is beyond the original description of an activity coordinator and the proposed activity-based system. In reality, this decreases the amount of handshaking necessary when two modules form their connection. Further, it decreases the creation of connections which are duplications, but would not be known until the new connection is made and the appropriate handshaking occurs. The module which performed the LOOKUP to the activity coordinator has knowledge of the role prior to any handshaking. This information is used for efficiency's sake.

3.3. Scenario

Let us look at how a task might work inside this framework. Assume there is an activity of maintaining a pig nursery that is composed of several activities; environment control, feeding, watering, and refuse disposal. We will look in particular at the task of environmental control. An activity is created and a module is started within the activity; the creation of a heating unit in the system starts the logical activity of environmental control. The object may request the start of other activities, or it may request that certain new objects become registered in this activity. In this latter case, the activity coordinator would send a request to the object implementor for the creation of the object, and upon receiving notice of these events, register the new object in the activity. This could be a creation of a fan within the activity. The fan then has the right to request further objects to be created in its activity (create another fan or heating unit), register other existing objects in its activity (register an existing thermometer) and/or to be registered in another activity (register itself in the activity of refuse disposal), while still registered in the environment control activity. This process may continue with the number of activities and objects growing and shrinking with time; of course, it could also remain constant. As objects finish their participation in an activity, they may request to be deregistered from that activity. The object may completely finish its participation in all activities and commit suicide. At this point, the coordinator will deregister the object and notify the controlling ACM of the event. The ACM will inform other objects in the activity of the death and, if desired, start another object. The affected objects may then request to be killed by the

implementor or request other objects to be killed. These contracts may be given out regardless of other objects' deaths.

3.4. Fault Tolerance Related to an Activity-Based System

A question of concern is, how does an activity-based system handle faults or facilitate the building of a fault tolerant system? An activity-based system does not handle failure of the system, in part or whole. However, the activity-based system will greatly increase the ability of a designer to build an application that contains elements of fault tolerance. First, we need to be concerned with the three basic elements for a software fault tolerant system: modularity, granularity and redundancy. Following this is a short explanation of how certain elements of dependability can be implemented with this system.

The language proposed for the system has only one built-in object type, the module. A module is a combination of data and code that communicates with the outside world by sending and receiving messages. Thus, the system expects modularity. Granularity of the developed system is left to the designer; the underlying architecture of the system enables the various objects and modules to be separate, both physically and logically, but to have a defined dependency through the activity. Finally, the designer is able to build as many activities and to define their dependencies. Thus, rather than having the implementor become concerned about whether the sites are separate and isolated, the main concern becomes the types of checks, comparisons, synchronization and validation. The requisite communication can be handled by the system. Any redundancy and use of that redundancy, for example in voting, are left to the developer.

3.5. Example

Assume that a paper company's assembly line for the production of folded, embossed napkin requires two robots, two conveyer belts, the folder/embosser, a box sealer and an operator's console to monitor the assembly line production. One robot is dedicated to maintaining the supply of paper to the folder/embosser. A second robot will remove the folded and bundled napkin from the folder and place it in boxes. When the box is full, it will place the box on a conveyer belt which runs to the sealer. The boxes will then continue, after being sealed, on another conveyer belt to the warehouse. For each action that occurs, the object managers send a message to the monitor and the monitor displays the action so that materials may be seen flowing from the front of the assembly line to completion (see figure 3.1).

One possible logical connection of all processes is the activity of making napkins for shipping, thus, one overall

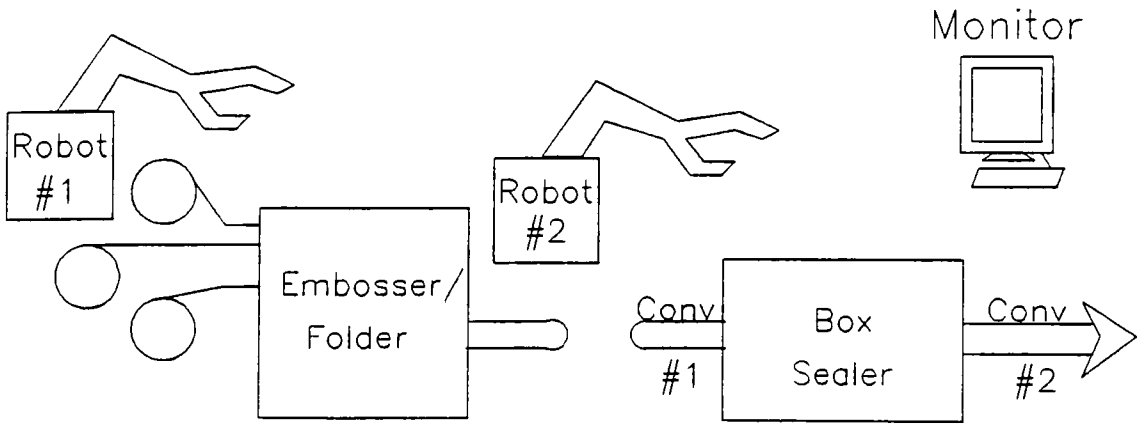

Fig. 3.1 Napkin Assembly Line

activity. This activity consists of the smaller activities of maintaining the paper flow, folding and boxing the napkins, and sealing the boxes for storage. The first robot and the folder/embosser (both managing the paper) belong to the first activity. The second activity of wrapping and boxing the napkins contains the folder/embosser, the second robot and the conveyer belt to the sealer. Another activity may be considered the conveyer belt to the sealer, the sealer and sending to storage (the second conveyer belt). Finally, for completeness, a fourth activity may consist of the last conveyer belt to storage, since there may be objects in the warehouse with which it may desire to communicate. The operator's console, or simply called the monitor, participates in each activity by displaying actions of each object within each activity as the modules send the information to the operator's console. Figure 3.2 shows a diagram of the logical connections of the system.

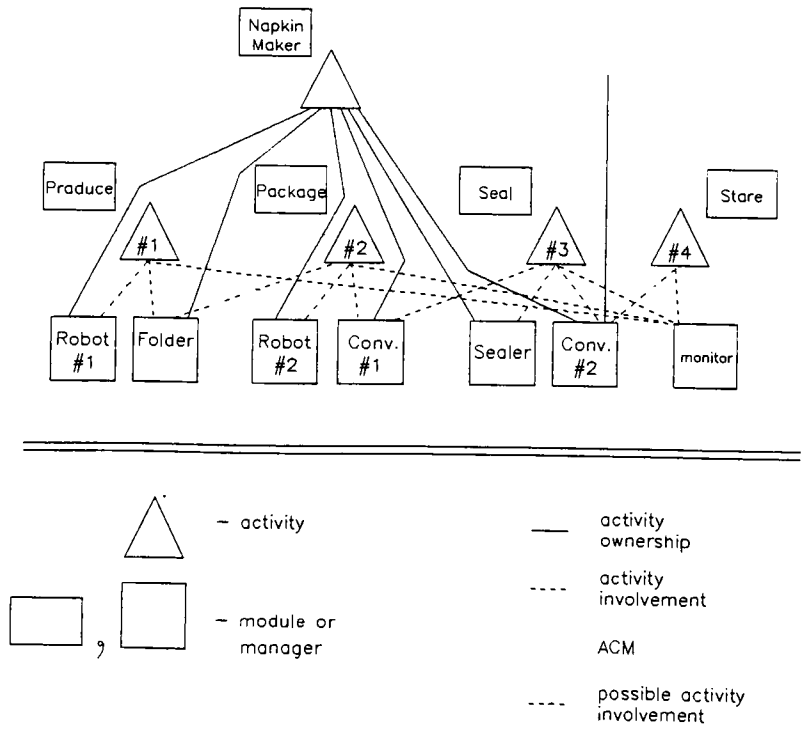

Fig. 3.2 Napkin Assembly Line Logical Organization

Now, one can see from the operation that to replace paper on the folder/embosser, the systems down the line should not have to halt their involvement in the system. As the paper runs low, the folder/embosser will send a message to the first robot about the situation. The paper will be loaded and production may continue. This down-time will eventually propagate through the system, but as a temporary slow-down and not a failure.

If the conveyer belt leading to the sealer fails, then production upstream should not be affected until the backlog becomes sufficient such that it is more prudent to halt production. The manager of the conveyer belt will announce the failure of its object. The manager may then attempt to reset and restart the conveyer belt. In the event it fails, there may be other options, but if all attempts fail, human intervention may be requested. Robot Two, having been informed of the failure of the conveyer belt, watches carefully the number of boxes stacking up. If a critical number is achieved, the robot halts and sends emergency information to the folder/embosser to halt production. This communication may take place via the activity control modules having the information of interested parties in the separate activities. For true fault tolerance, it may be prudent to maintain a second conveyer belt for the sealing and storage but not for the folder/embosser.

This is not an exhaustive search of all possible failures. Yet, I feel that this example shows the ability of the activity based system to handle the heavy requirements of communication and allow flexibility of components required with fault tolerance.

A significant factor, not yet expressed, is that this system involves a real-time application. Because it is event-driven, not clock-driven, aids in this factor;

therefore, we do not need to be concerned about absolute timing but relative. This system is typical of assembly line manufacturing and makes it very applicable for industry.

4. IMPLEMENTATION

For my thesis, I have implemented one aspect of this system, the module implementor. The module implementor, the activity coordinator, and the simulated system were developed on diskless SUN 3/60 workstations under SUN UNIX rev. 3.4 with a network file server. Aspects of design and implementation which are dependent upon this are noted. In this chapter, I will discuss the design and capabilities of the module implementor.

4.1. Command Capabilities

The implementor is able to execute many commands, as listed in Appendix A. Except where noted, the return value is either an exception or successful return, and it is at this point where we may determine and expect atomic actions. All requests occur via network communications and messages. The proper format of each request is shown.

4.2. Design

The design of the implementor must be able to handle failure of itself, components of itself, or of modules executing under its control. Thus, the implementor is composed of three sections: the listener, the main section and watchers.

The listener will reside at a well-known address forever. It currently has two purposes. Its primary purpose is to forward messages to the main section. If the main section has died, messages would be sent back with this information. A main section death currently demands user

intervention. But, it would be a simple matter to enable the listener to restart the main section (see Chapter 6).

The main section will receive the commands, spawn watchers and manage the watchers (see Figure 4.1). The watcher is explained below, but for each new module there is a watcher. The main section is able to watch over the other sections of the system, thereby guaranteeing stability.

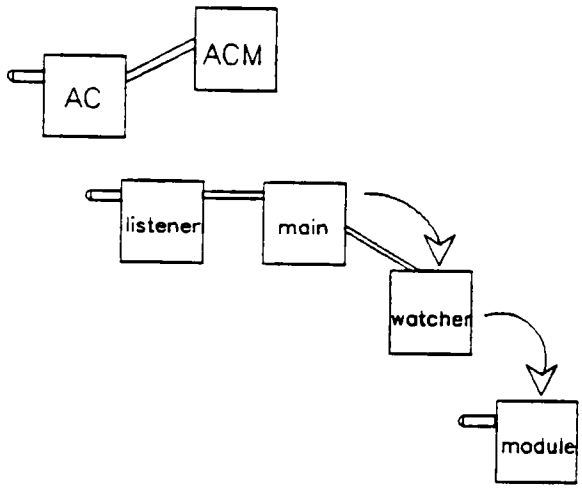

Fig. 4 1 Creation of a Module

The watcher's sole purpose is to watch the module. If a module fails, the main section and the activity coordinator are informed, and the watcher dies off. If a new watcher is to be created, the main section of the implementor receives the request and starts a new watcher, which actually begins execution of the module. The watcher also handles the "warning time" notice for destroy commands. The watcher knows the module is to exit, and if it does not, kills the module and tells the main section.

This design was adopted with the expectation of utilizing certain characteristics of the UNIX operating system, in particular : ability to "fork" and "exec"

processes, socket communication, software signals, process groups, and death notices.

A primary functionality utilized are the "fork" and "exec" system calls. A UNIX process may start a separate process which uses a separate code path in a copy of the original program file, using the "fork" system call. The forked process will be executing a copy of the original program file but will not return within the same context as the originating process. This second process is then able to load and execute another program file using the "exec" system call. Thus, the implementor would be able to start as many watchers as would be needed. In like fashion, the watchers would then be able to start the execution of the various modules. Further, the forked and exec'ed process inherits the open file descriptors of its parent process.

Unfortunately, software signal handlers are not maintained after an exec call or this functionality, too, might have been utilized. The implementor, as the ultimate descendant, would have been able to set signal handlers for the watchers and modules even before the modules were created. But, because of this descendancy, the implementor is able to send software signals to the watcher and their children, the modules, asynchronous to the execution of the processes. Therefore, when one module desires to connect to another module, this mechanism could be used. For asynchronous actions such as killing/warning a module or connecting to another, the implementor sends a software signal to the process group that the watcher created (consisting of the watcher, the module and any children of the module). The watcher, upon receiving a warning/kill process, sleeps for a period of time and kills the module if it has not already completed execution. The modules may use this period of time to prepare for death or ignore the

information and continue the execution until it is killed. The watcher, after creating the module, is able to devote its time to waiting for a death notice from the operating system, evaluate that information, and return the evaluation to the implementor and to the coordinator.

I believe this design gives the highest degree of modularity for the different aspects of the system; at the same time, it enables proper monitoring of the modules. Further, it spreads the requisite functionality throughout its various components. This modularity definitely enhances the future possibility of system improvement discussed in Chapter 6.

4.3. Coordinator

An activity coordinator is responsible for maintaining the various components of the system as well as registration lists. This implies that activity-related actions occur via the coordinator. Therefore, module creation has been implemented to happen through the coordinator. The coordinator also receives notices of module death, informs other modules of the death and acts as a user resource for activity information.

Any decisions that need to be made upon receiving activity information would, eventually, be sent to an Activity Control Module, that is tightly coupled with the coordinator. The coordinator for this specialized system acts in its place.

The coordinator's main function in this specialized system is to act as an intermediary for module creation, maintain system registration lists, and upon receiving death notices, inform interested modules.

4.4. Command Execution

For several of the commands, an explanation of the internals is required. Further, several of these commands may in fact be a series of commands at the lowest level. For this reason, I will give a process and connection level explanation of the processes involved.

The Activity-Based System is configured to have one central Activity Coordinator (in my implementation) and a listener and main processes on each station in the total system (see Fig. 4.2).

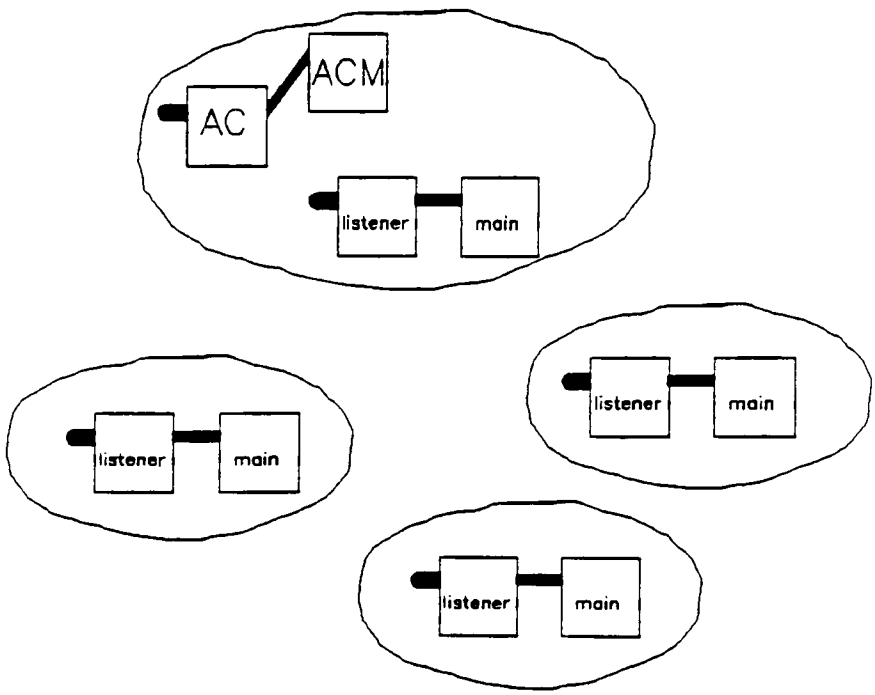

Fig. 4.2 System at Start

4.4.1. Connection of a Module to an AC

The address of the Activity Coordinator is well known and therefore readily available to all processes on all systems. This address must be known within the context of the UNIX operating system. There is no capability to determine dynamically the address of a server process, a main impetus of this thesis. To connect to the AC, a simple connection needs to occur (see Figure 4.3). This information is presented for completeness.

4.4.2. Module to Module Connection

In the lowest level of the system, when a module requests to be connected to the other module, the modules must first determine the locations of the other modules. To accomplish this, the module may connect to the AC and execute a LOOKUP. The return information is the complete network address and the role of the modules within the activity. The module may then connect to any particular module or to all modules in the activity.

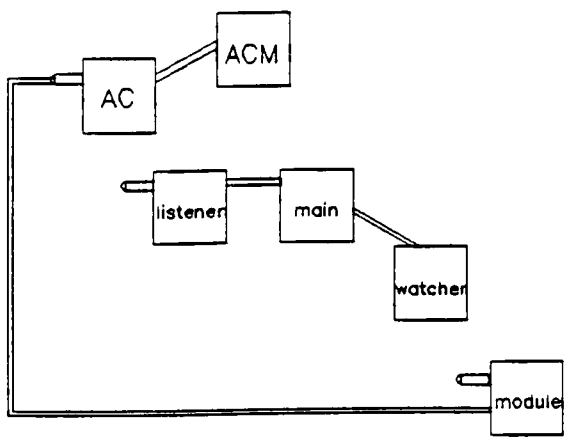

Fig. 4.3 Connection of a Module to the AC

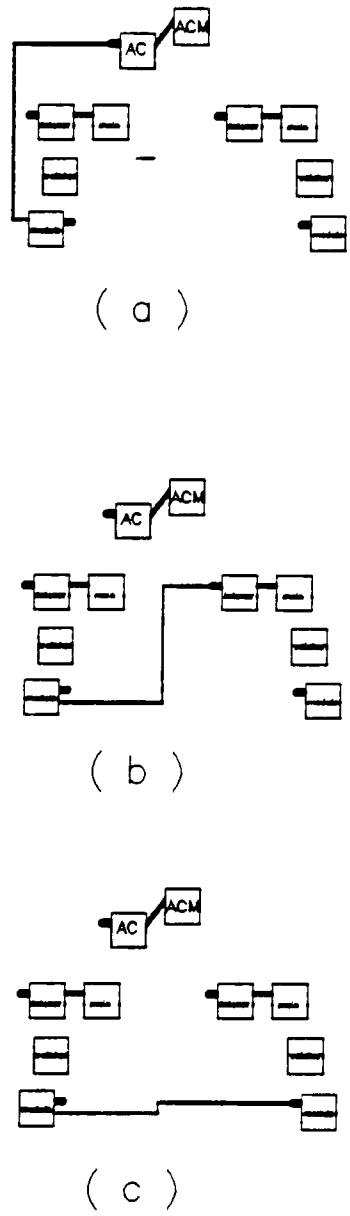

Fig 4.4 (a) LOOKUP (b) CONNECT (c) Module Connection

For a module to connect to another module once the address is known, the requesting module must first request from its module implementor a BLOCK. If there is an outstanding connection request to this module, the request will be refused. Once a successful BLOCK has been performed, the module must connect to the implementor of the target module to request a connection. If the target module has previously requested a BLOCK or another module has an outstanding connection request, this request will be refused (see BLOCK and UNBLOCK in Appendix A). If neither condition is true, the implementor then signals the destination module of this request. Upon receipt of the successful connection request, the module may connect to the target module. Once the conection is completed, the requesting module must reconnect to the target module's implementor and inform it of the completed connection. Finally, the module must reconnect to its implementor to UNBLOCK itself so that other modules will be able to connect to it (see Figures 4.4. a,b & c).

These actual steps are hidden from the casual user, and there are layers of calls or library routines that the user may use so that these steps are completely transparent (see Appendix C).

5. Simulation

To verify the capabilities of the system and, more explicitly, the implementor, the example of the assembly line given in Section 3.3.1. was simulated. The system consists of seven collections of code:

1) Robot #1
2) The Embosser/Folder
3) Robot #2
4) Conveyer Belt #1
5) Box Sealer
6) Conveyer Belt #2
7) Operator's console

Each component of the simulation was implemented with one module representing an implementor/manager. Each module forks another UNIX process representing the simulated managed object. The simulated objects receive their commands from the manager. Upon receiving a command, the object reads from a file if the particular command is to succeed or fail. This is to simulate the actual failure of an object for the manager and the overall results on the system.

To start the system, a module independent from the simulation connects to the Activity Coordinator to request the creation of the operator's console in activity one. This independent module, having completed its sole purpose, dies off.

5.1. System Related Routines

The explicit format and information to call these routines are in Appendix C. It was assumed that future developers utilizing this system would desire to use many of the previously implemented routines. Therefore, the developer need only include the file "activity.h" and link their files with the library file created for that particular machine. Items that need to be modified for migration to another system are well documented in the included files and need only occur at that level. An example application is described and system modifications that need to occur are described in Appendix D. The final library object file may be obtained by using different switches on the compile argument line.

5.1.1. Initialization

Upon creation of each module, the module goes through an initialization sequence to create the assumed execution environment. This routine utilizes two pieces of information: the activity in which the module was created and the role of the module. The former information is passed on the argument line by the activity system. The latter is known by the programmer at the time of implementation.

The main purpose of this initialization routine is to connect the module to the other modules within the activity. This is accomplished by connecting to the activity coordinator and requesting a LOOKUP on the activity (total functionality of this function is described below). For each module in the activity, if there is not an existing communication connection, the module would connect to the module (the two sides of the connection mechanism are discussed below).

In this mechanism, the communication is maintained nearly virtual with possible performance degradation. The need for the module to be knowledgeable about and execute system calls is therefore deemphasized. In addition, there was no need to be concerned with the actual order of the creations. If two modules have a need to communicate and the second module is not yet created at the time of the LOOKUP of the first, the latter module will receive the information of the first module and a connection will be made at that time.

This routine also initializes the connection handlers for a software signal. The need for this is discussed below.

5.1.2. Software Signals

When an object connects to the implementor to request a connection, a software signal is sent to the module. This mechanism is used due to its capability to occur asynchronous to the execution of the target module. The connection handler can handle two different types of requests: death notice and connection request.

The former command comes from the activity coordinator to inform the module of a death of a module in an activity in which the module is registered. This command causes the module to make the file descriptor invalid for future use. Therefore, before a write to a module, the module can verify that it still has valid communication.

The connection request between modules also contains the role of the requestor. Both modules verify that they do not have an existing connection and that this is to be the connection between the two. This file descriptor is then set into a socket array using as the offset the role of the requestor.

When a module is to be brought down, the kill request is sent to the implementor of the module. This causes the

implementor to send a software signal to the module. There is no default handler for this signal. This is to set up a warning routine so that the module may shut down gracefully. The warning handler for the embosser/folder determines if it has sufficient paper to justify restarting the machine with that amount of paper. If there is less than this amount, the embosser/folder will continue executing to hopefully finish its current material. The monitor, upon receiving a warning signal, simply completes its execution. It has no material with which to be concerned. All other modules will ignore the warning and will continue to process their material until they are killed by the watcher.

5.1.3. Register

The register routine takes three arguments: the system id, the activity number in which the module desires to be registered and the role of the module. The id and new activity are sent to the activity coordinator to be registered. A LOOKUP is then performed by this routine for all modules in the new activity, and connections to currently unconnected modules are performed.

5.1.4. Deregister

Two arguments are required by the deregister command: the system identifier and the activity number which the module wishes to discontinue interaction. This routine receives from the DEREGISTER command from the activity coordinator the identifier and role of the modules which are currenlty registered in this activity. These modules for which the information is returned are not currently registered in another activity in which the this module is participating. This routine then uses this information to close the sockets which are only related to this specified

activity. The activity coordinator will inform the interested objects of the deregistration of the module so those objects may close the apropriate socket.

5.1.5. Lookup

To perform a LOOKUP, retrieve connection information about all modules registered in an activity, the module connects to the activity coordinator and sends the number of the activity in question. The activity coordinator returns, for each module in the activity, the network address of the station on which it is executing, the socket port number, the system id and the role of the module. The coordinator will close the connection when it has finished with all modules in the activity.

5.1.6. Connection

When a module in the simulation determines that it does not have a connection to a certain module and desires to set up a connection, the module must first connect to its implementor and request to BLOCK connection requests. This is performed so that while the module is connecting to another module the connection process is guaranteed to be atomic. If this does not occur, a deadlock situation may arise among connecting modules. When this permission has been received, the module must then connect to the implementor of the target module and request a connection. If the target module is BLOCKed, the implementor will refuse the connection, the requesting module will UNBLOCK incoming connection requests and time out a random period of time. This continues until permission is granted by both implementors. In granting permission to the requesting module, the implementor of the target module sends a software signal to the target module and blocks any further connection

or block requests from succeeding. The requestor may then connect to the target module. There is a possibility that the target module has connected to the requestor. Therefore, the requestor tells the target whether it does or does not currently have a connection with this module. The target module will verify this status and return agreement to the requestor. At this point, the requestor and the target agree that this socket is to be their channel for communication. Both modules use the role of the other as an offset into an array of file descriptors and set this value to be the value of the new socket. The target module will continue with its previous execution. The requestor must then connect to the implementor of the target module, informing it that the connection is completed so the target module is able to receive other connections and to BLOCK itself. Finally, the requestor connects to its implementor and requests an UNBLOCK, making itself available to other connection requests.

5.2. Module Actions

I will hereafter describe the actions of each individual component of the simulation. Many of the actions occur concurrently. All modules start execution by performing the initialization mentioned above. Upon completion of specific registrations and creations, the modules start their work interactions. The actions of the modules are discussed below in greater detail. The complete overview of activity-related events is shown sequentially and concurrently in Table 5.1.

initial process – CREATE Folder CREATE Monitor

Monitor	Robot 1	Folder	Robot 2	Conv. 1	Sealer	Conv. 2
INITIALIZE Activity 1		INITIALIZE Activity 1				
REGISTER Activity 2	INITIALIZE Activity 1	CREATE Robot 1				
REGISTER Activity 3	WORK	REGISTER Activity 2				
REGISTER Activity 4		CREATE Robot 2				
WORK		WORK	INITIALIZE Activity 2			
			CREATE Conv. 1			
			WORK	INITIALIZE Activity 2		
				REGISTER Activity 3		
				CREATE Sealer	INITIALIZE Activity 3	
				WORK	CREATE Conv. 2	
					WORK	INITIALIZE Activity 3
						REGISTER Activity 4
						WORK

Table 5.1 Simulation System Activities

5.2.1. Robot #1

Robot #1 does not request the creation of any other objects. Further, it behaves differently than the other modules by not having the possibility of continuous work. Robot #1 pends execution waiting for a request for paper. Upon receiving this request, the command is sent to the actual machine. A successful load returns to the manager the weight of the paper that was loaded. The module sends this information to the requestor and waits until it receives another request.

5.2.2. Embosser/Folder

The embosser/folder is created in the context of activity one and requests the creation of Robot #1 in this context. It then registers in activity two and requests the creation of Robot #2 in this new context. The embosser/folder then begins the task of paper management and napkin creation.

5.2.3. Robot #2

Robot #2 requests the creation of Conveyer #1 and begins the process of receiving napkins to be boxed, boxing them, and sending them to Conveyer #1.

5.2.4. Conveyer #1

After registering in activity 3, Conveyer #1 requests the creation of the Sealer. It then starts its main task of shipping boxes to the Sealer.

5.2.5. Sealer

The Sealer, prior to starting the sealing of boxes and giving them to Conveyer #2, requests the creation of Conveyer #2.

5.2.6. Conveyer #2

With the possibility that there is an activity 4 controlling paper storage in the warehouse, Conveyer #2 registers itself in activity 4. It then starts its main work task.

5.3. Module Work

The modules operate with very similar logic. The premise is that the machine will continue to work as long as it has sufficient material for its task; therefore, this is the default operation. The commands that the different managers accept differ slightly but may be generically categorized as YOUR_WORK. The modules are able to accept the commands: HOLD_UP, CONTINUE, YOUR_WORK, INCREMENT_AMOUNT, DECREMENT_AMOUNT, INCREMENT_TIME, DECREMENT_TIME and HALT. The HOLD_UP command comes from the object "down-line". This command sends the module into a state such that it accepts material from other objects but does not send it on. If the module receives too much material for it to contain when it is in this state, it may send this same command "up-line". When the problem "down-line" is rectified, it will send a CONTINUE command. This informs the machine that it may continue processing as normal. When its supply of material is low enough, it will send the "CONTINUE" command "up-line" to the previous machine if it has previously informed that machine to HOLD_UP. The parameter modification commands (INCREMENT_TIME, DECREMENT_TIME, INCREMENT_AMOUNT, DECREMENT_AMOUNT - shown in Figure 5.2 as PARAM_MOD) give the capability to modify the amount of time it takes to produce a particular amount of work and the amount of work that may be produced for the duration of time. These commands give the capability to modify production rates to verify back-ups and communications in general. The commands originate from the operator's console which may send a specific command to a specific machine. When all execution is to be stopped in an orderly fashion, the command HALT should be sent to the first machine in the assembly line. This will cause the machine to process all the material that it has and forward the HALT

command to the next machine in the line. This command may also originate from the operator's console. A pseudo-C code of the modules is shown in Figure 5.1. and a state diagram is shown in Figure 5.2.

The exceptions to this generic execution are the operator's monitor and the embosser/folder. The monitor execution was previously stated in section 5.2.7. The embosser/folder must also check the amount of material it has available to produce napkins. If there is insufficient material, it sends a request to Robot #1 to add paper.

```
forever {
    if ( NEW_COMMAND )
        to_do = NEW_COMMAND;
    else
        to_do = YOUR_WORK;

    switch( to_do ) {

        case CONTINUE :
            state = NORMAL_OPERATIONS
            break;

        case HOLD_UP :
            state = RECEIVE_ONLY;
            break;

        case YOUR_WORK :
            if ( state == RECEIVE_ONLY ) {
                accept_new_material();
                if ( amount_material >= MAX_BACKUP ) {
                    send_previous_machine(HOLD_UP);
                    prev_mach_state = HOLD_UP;
                }
            }
            if ( state == NORMAL_OPERATIONS ) {
                if ( tell_machine_to_do_work(time, amount) )
                    send_next_machn(YOUR_WORK,produced);
            }
            if ( ( prev_mach_state == HOLD_UP ) &&
                ( amount_material <= OK_LEVEL ) ) {
                send_previous_machine(CONTINUE);
                prev_mach_state = NORMAL_OPERATIONS;
            }
            break;

        case HALT :
            if ( state == NORMAL_OPERATIONS ) {
                while( amount_material >= ENOUGH_TO_DO_WORK ){
                    if ( tell_machine_to_do_work(time, amount) )
                        send_next_machn(YOUR_WORK,produced);
                }
            }
            send_next_machn(HALT);
            commit_suicide();
            break;
```

Fig. 5.1 Pseudo-C Code of Module Execution

```
        case INCREMENT_TIME:
            time += TIME_UNIT;
            break;

        case DECREMENT_TIME:
            if (time > TIME_UNIT)
                time -= TIME_UNIT;
            break;

        case INCREMENT_AMOUNT:
            amount += AMOUNT_UNIT;
            break;

        case DECREMENT_AMOUNT:
            if (amount > AMOUNT_UNIT)
                amount -= AMOUNT_UNIT;
            break;

        default :
            commit_suicide();
            break;
    }
}
```

Fig. 5.1(continued) Pseudo-C Code of Module Execution.

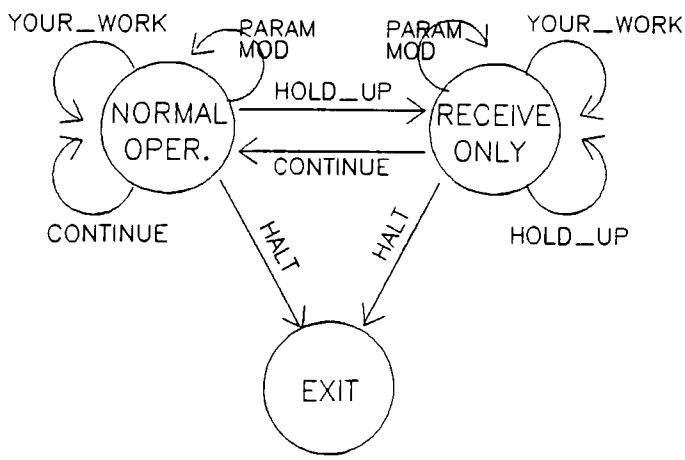

Fig. 5.2 State Diagram of Module Execution

5.4. Final System

The complete system is pictured in Figure 5.3 with all the modules created and the module to module connections existing. The front (robot #1 and folder/embosser) part of the assembly line may have started execution before the full system is completely configured, but the remainder of the system should be finished with its set-up before any communication propagates through the system.

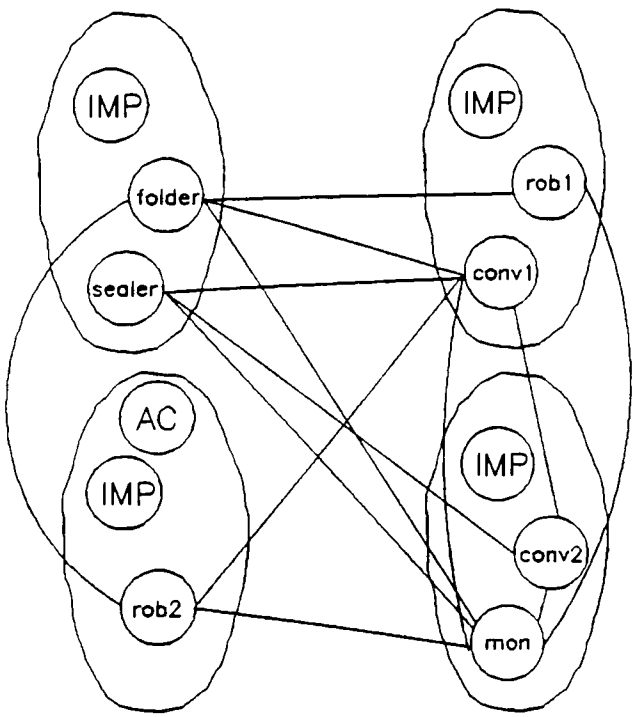

Fig. 5.3 Total Simulated System

6. Implementation Issues

During the development cycle, in particular the implementation stage, there were several difficulties related to the implementation of the original specifications. These difficulties were a result of the UNIX environment, an attempt to keep the complexity of the implementation minimal, a desire to minimize the overhead of the system, and lack of time available to do the type of implementation which is now envisioned.

These areas of deviation are emergency notices, fault tolerance, process signal state, and the executing environment.

6.1. Emergency Notices

In certain instances, the module implementor, activity control module, the "watcher", or the module itself needs to send and/or receive emergency notices to inform/learn about important activity phenomena, such as module or object death. For the reception of these high priority messages, in UNIX, the "out of band data" must be used. This causes all queued data on the data stream to be flushed up to a "out of band data mark". One would then have no method to determine what information was lost. All senders of the information would have to know that this could occur, and resend the information. This would demand very repetitive and, thus, slow overall response.

Had the design been more thorough, this phenomenon could have been more transparent to the user and acceptably implemented. The user overhead which would result currently to accept the inevitable loss of information is definitely unacceptable.

6.2. Fault Tolerance

In the original thesis proposal, a great deal of discussion was given to the area of fault tolerance. Different methods and levels of fault tolerance were considered. It was determined that asynchronous snapshots could be used. Major modifications to copy user and system stack information and other relevant process information would need to be done. This information could be stored to long term storage (tape or disk) and used to walk-back to a previous system state.

Several main issues arise. The implementation may be difficult in a "real" environment where objects and modules may be irretrievable, i.e. an item has already gone off the assembly line. In this environment, one may desire to shut all systems (nodes and peripherals) down and have an operator intervene before restarting the system.

Further, the overhead incurred may not be useful for the purpose of this simulation. This implementation is aimed more at the practicality and reality of an activity based system. At some future point, some level of fault tolerance may be added to enhance the flexibility of the simulation. Any application executing in this system may, of course, use the flexibility and characteristics to build a more fault tolerant system. It is a combination of these factors that lead to the decision to not include fault tolerance as part of the simulated system.

An emphasis of an activity based system is that logically associated objects receive information about the death and existence of other objects in the same activity. Thus, a module within the activity would be able to request the creation of a module or object after its death. Or, the activity coordinator under the guidance of the activity

control module would be able to automatically request the recreation of the object. With this as a premise, the implementor of the modules would be able to save information at logical points within the module and regain that state winthin the reborn module. Other modules would be informed of the death, wait until it is recreated and have a high level of confidence in the state of the recreated module.

A more complete system could include a check-point logger of state information from the objects. Upon the death of an object, this fault manager could inform the other modules of a previous state that all modules should back up to and wait for the recreation of the dead object.

6.3. Signal State

The state of a module and a node is not implemented well. In both cases, if it were re-implemented, modifications should be done to UNIX and the implementor.

In the former case, UNIX does not inform the sender of a signal about the process state of the signal target process. The return value informs the sender that it had the privilege to send the signal to that process, or the target process was found. This lack of information resulted in the implementor being modified by putting in the receive message state of a process. Modification should be done to UNIX to return information such as the target process ACCEPTED, BLOCKED, or IGNORED the signal. This is not difficult to do. Once the process table is found for a process, the signal may be placed in the the appropriate offset. Then, the operating system may determine if a handler has been installed for the signal value (offset in an array in the process table) and return the state of the target process to the sender.

In the case of node state, information as to the network accessibility of a node should be kept either internally by each implementor and/or by the activity coordinator. This would increase the efficiency of the system considerably. Rather than attempting a known impossible request, the activity coordinator and/or implementor will return a failure immediately. Further, a boot count would be useful to determine if a node went off the network or became temporarily unavailable and the station involved is oblivious to this fact.

These phenomena can be worked around. But, future revisions and enhancements should take them into consideration.

6.4. Module Environment

In UNIX, by utilizing one of several system calls, a user is able to start another process. Unfortunately, the user is not able to explicitly define the environment in which this process will execute. Due to this inflexibility, an explanation of the executing environment is needed, in particular, the message command line, default channels and connectability of modules.

6.4.1. Command Line

In a message sent to the implementor or to the implemented activity control module, due to lack of forethought, no arguments may be passed from the command line. This is a minor inconvenience, and control files may be hard-coded into the program for parameter information to be used by the system.

6.4.2. Communication Channels

At normal process initialization, each process is given three default open channels: standard in, standard out, and standard error. In this environment, these channels are not useful. Except in the case of standard error, to send error messages to a control console or monitoring device, the modules want to manipulate special peripheral devices such as machinery. Further, since network communication occurs via the same mechanism (a file descriptor) and interprocess communication is a standard element of the activity based system, each module should receive an open network address socket. Therefore, the only default open channels should be standard error and standard socket. All other peripherals, files and other types of I/O should be opened explicitly within the module.

During the course of the implementation, there was a noted deficiency in the capabilities of the read/write functionality. Upon a write call, UNIX does return the number of bytes written or error condition. This is not to say the information was received or that the process on the other end of the socket is still there. The file descriptor is still valid on that end. Reading to this file descriptor will return an empty string. This does not necessarily represent process death, but there are not many other conclusions that can be presented. Further, a "select" on a file descriptor when the other process is dead will return that file descriptor as valid for a read. These conditions caused some modification to the simulation code in assumptions about module death, even though the coordinator had not yet informed the module of the death. Another modification was the need for greater handshaking between communicating parties.

6.4.3. Connectivity

In UNIX, it is a documented feature ([UNI83], signal(3)) that if an I/O request is being executed by a process when a signal is received by the same process, the I/O request is halted and the corresponding exception is returned to the requester. The documentation claims that the I/O request is retried. I have not found this to be true. For all I/O requests, the error return must be checked, and if it is the aborted request exception, the request must be redone.

When a module issues a connect request to another module, the system determines the location of the module and sends a signal to the target module (if it is in an accept connection state). This, unfortunately, means that all I/O requests must be coded for the exception case.

This feature is scheduled to be fixed in a future revision of UNIX.

6.5. Product Readiness

There are many areas in which existing code could be improved using the experiences of the operating system and characteristics of a distributed system. As a "real" product, this system in its current implementation would be deficient. Yet, this "product" is also just starting its maturity stage. The trade-offs for a truly fault tolerant distributed system are numerous, and must occur at some time in its development cycle.

7. Issues and Related Topics

During the research, implementation and writing of this thesis, several issues and related topics were discovered. These areas would be possible future thesis topics related to an activity based system, distribution, and computer science in general.

7.1. Activity-Based System

Many areas of the activity-based system have yet to be developed, including activity coordinators, activity control modules, object managers and implementors and migration of the system into the operating system.

The activity coordinators and the activity control modules may be tightly coupled components, but the need for this tightly coupling needs to be investigated. Regardless, both components need to be implemented, including many concepts in the remainder of this chapter. This would involve more precepts of distribution and implementation of a distributed system.

Investigation into types and classes of object managers and implementors may be enlightening regardless of specific distribution and communication characteristics which may be determined. Classifications of managers and implementors may continue to be a concern for non-distributed stand alone environments. Therefore, a better understanding of types could improve designs of these components.

A final goal of accumulated efforts is an activity-based environment with those characteristics defined. Code written for this environment would be most efficiently written and executed if it were compiled by a compiler designed and implemented specifically for this environment [HEL84]. Not

only will this area require knowlegeable compiler construction but add the characteristics of a compiler targeted for a distributed environment.

Finally, many aspects of this system may be included within an operating system. By becoming "trusted code", the activity-based components may be able to decrease memory usage, utilize system primitives and improve performance.

7.2. Fault Tolerance

The full topic and many aspects of fault tolerance are too numerous to list completely but there are several parts related to an activity based system. These areas include dynamic redundancy, snapshots, and voting.

In the area of dynamic redundancy, the possibility of having dynamic reconfiguration almost virtual to the user, redundant peripheral communication or control, system redundancy in the area of hardware fault or error would be particularly useful for a distributed environment and for an activity based system. Modules or objects might be migrated or started on another node using the mechanisms and information available in an activity based system.

The area of snapshots is greatly disputed and fairly well documented. Systems that implement, or theories that describe snapshots are areas for future theses of general interest. A distributed snapshot may also be of great interest as the computer industry utilizes more network communication and distributed systems. Targets for an activity based system could include complete or individual system snapshots to enable the systems to walk-back to a previous system state.

Finally, the many aspects of group decisions might be developed. For example, if the activity coordinator or part of the activity coordinator cannot be reached via the network (either due to node death or network discontinuity), does the system shut down or start another? Can modules be relocated to a different node? If so, does this mean there are different types of modules and objects?

7.3. Performance

The ability of the system to react to "real" peripherals is untested. Further, no figures exist as to the latency of user requests. The figures of performance were not an issue. The capability to implement an activity based system was the goal.

Since this has been achieved, in part, the next step would be to re-code or redesign certain areas of the system so that latencies are minimized and at least measured. A limiting factor of the current design is that the system does not exist as part of UNIX, and that UNIX is not a real time operating system.

Upon determination of these last two points and where the system is going, further design modifications should be made and the cycle restarted.

7.4. Guaranteed Communication

As mentioned in Chapter 6, Implementation Issues, there are some deficiencies for guaranteed communications between processes, particularly between remote processes. This could be modified by some handshaking between servers within UNIX so that the read/write return exception cases if one end of the communication socket disappears. When the information

has been written to or read from a remote station, that station would be able to return exception if or when the process dies. Of course, for a write, this still could not guarantee that the buffer has been read before a process death. Of course, there are no 100 percent guaranteed network communication theories, but it seems there could be a great deal of improvement of the current level of confidence between remote systems.

7.5. Process Hierarchy

I have personally worked with two forms of traditional process hierarchical tree stuctures in distributed systems. In the first, a remote process receives a process id (pid) within the hierarchical structure of that operating system. All processes are descendants of an ultimate process. In another approach, there could be a disjoint process tree. Remote processes are descendants of the originating system and are within that process tree structure. A third approch is that of the activity-based system. There is no direct descendancy implied by the pid or locality of the object, necessarily.

I would argue that each of the approaches have benefits and deficiencies. Further, the traditional approaches should be re-investigated, to determine how well they lend themselves to an interactive distributed computing environment.

7.6. Debugger

During the course of implementation, a great deal of debugging was required. As a result, a need for more substantial debuggers was determined. The SUN workstations

which were used for the implementation have a nice facility to map in executing processes by passing the pid of the process to the debugger. Without this facility, the difficulty of this implemenetation cycle would have been greatly increased.

Some possible enhancements, which currently are not implemented, follow. When a process is executing in the debugger, the process is unable to execute a "fork" command in the debugger. There is some sort of difficulty with addressing, and execution in the debugger cannot continue. I would think that the debugger should be able to follow at least the father process and let the child continue on. A remote process cannot be debugged. I realize the difficulties with addressing that are involved but have seen a need for this functionality in a distributed environment. The reception of software signals could be improved. Since this is supposed to (possibly) cause an asynchronous execution of the process, it should just execute the signal handler. If the person debugging the process desires to follow the execution, breakpoints could be set in the routine. You cannot have mutiple debugging processes of a single process. This, again, causes addressing problems.

Finally, it would be nice to be able to map in the address space of the executing process and be able to execute a "shell" or "macro" program on the data structures and information contained within the process. This would be particularly useful for the debugging of the operating system or very large processes.

These are a small sample of thoughts on improvements to existing debuggers with which I have worked. I am sure there are many other elements and issues that could be investigated in this area.

8. Conclusion

When starting the system, the module implementor comes up and opens its network socket, waiting for the first request. The activity coordinator acts in kind. The starter process requests the creation of the operator's console. After creating the embosser/folder from the operator's console, all other modules are created, registered and connected in their activities of participation.

Unfortunately, the simulation did not quite behave as it was perceived it would. The modification of the parameters to some extent affected the results within the system for hold-ups and continued production. Yet, these hold-ups could only be reached by increasing the amount "upline" by a great deal and decreasing the amount those objects "downline" could produce. The implementation of the object manager did not allow sufficient concurrent execution between reception of the request/material and the production done by that object. Therefore, the system simulated an unbounded backlog of materials enroute to a particular object.

Further, the simulation was insufficient in its reaction to object death. Once the object had received the death notice, the connection was closed. This should have sent the module into a RECEIVE_ONLY state. The death notice not only prevented the object from sending material to the next object, but since it pended on the recreation of the object, it prevented the object from receiving more material. Thus, an unlimited backlog was created. The premise of quick recreation of a dead object was erroneous. Another affect of closing the connection upon reception of a death notice, such as received when the machine was HALTed, was outstanding material to be manufactured was lost. In similar fashion, a final HALT command to propogate through the system may have

been lost. In this case, each machine had to be HALTed individually.

The activity coordinator and the activity control module (implemented as one), are insufficient for a true system. It would be possible, however, to implement an actual system or another simulation with the implemented coordinator. There are also insufficient dynamic capabilities and intelligence in the coordinator for common usage. A current developer would need internal knowledge of the coordinator to accomplish a true working system.

I have come to believe the activity control module and activity coordinator may be the most important aspects of producing a truly dynamic flexible activity-based system. Further, not only could a flexible activity-based system be developed with a singular coordinator, but it is possible to implement a realistic distributed coordinator with corresponding control modules.

The module implementor is sufficient and capable in its role. As mentioned above, the simulation did not react intelligently to the reception of the death notices. Yet, it was sufficient to prove that the full functionality of the module implementor exists in the implemented module implementor. The commands for blocking and unblocking were added in hindsight upon working with the simulation and incurring deadlock between several modules. The original conceptual designs of the module implementor were optimistic and too elementary for a working module implementor. This further insight into the activity-based system also required modifications to some of the user library routines. I believe the simulated system tested its capabilities, and the module implementor is on the path to becoming a mature software product. The next step or cycle of development for the module implementor would include greater parallelism and

all of the associated issues. Further, the code should be moved into the operating system. This would increase the possibility of atomicity and improve performance.

I found many weaknesses in utilizing the existing network protocols and retrofitting its capabilities into the needs of the activity-based system. By utilizing raw sockets and building a layer for the user interface, there could be a greater level of confidence in the communications and in the atomicity of the communications. If a module died, the coordinator was informed and in like informed the requisite objects. However, I found a considerable window in which a module could send information to the dead module with no error returned before the sending module is informed that the target module is dead. By moving the module implementor into the operating system, allowing the implementor to participate in death notifications, and utilizing a different network interface, this window could be closed.

Performance is a main issue in much of the above discussion. I did not take actual timings of the system. I felt this would not have been useful. One reason for this is I found the simulation started and executed differently depending upon the load on the network. Since the activity-based system and simulated system were implemented on diskless SUN workstations with a network file server, network traffic created a large fluctuation in the perfomance of commands, such as create, register, deregister, and death notices. Further, the activity coordinator needs improvement of its design, and many of the commands would be measureing the coordinator. Finally, true timings of the system would require building special simulations to do the timings. For example, to measure the timing for a register command (a default component of the create command), there are many interwoven issues and different possibilities: the number of

required module connections to be made, which is dependent upon previous connections, the number of modules currently registered in the activity, whether other modules attempting to perform connections, and whether that module is in the same or different activity as the module which is being timed. Thus, any timing figures may not really have given any useful information and may have been misleading.

Another important issue previously mentioned is atomicity. We have seen from the Argus system that the information and capabilities it offers cannot truly be guaranteed, and the needs of a distributed computational system are more than mere atomicity. The activity-based distributed system offers more flexibility and information and thus allows for more evaluation to be made by the user. Yet, network failure and characteristics will continue to be an important issue to be resolved. If you cannot communicate to a system, does this imply that the system and all associated objects/modules have died? Thus, even with atomicity, you may know the existing state but when the system cannot be reached can the current state be determined or guaranteed.

I have developed distributed applications under a standard operating system environment (UNIX), under a distributed environment at my job, and under the activity-based environment. I feel that, of the three, the activity-based system is the best environment in which to develop a distributed application. The activity-based system offered the most information and required the least amount of process communication management. These areas are core elements to the design and implementation of a distributed application.

I am sure further investigations will be made into distributed processing. The classic executing environments

for computational systems are not sufficent to expedite the growth of distributed computing. There will be numerous attempts at the "best" environment before a final usable solution has evolved. I believe that environment will maintain many characteristics of an activity-based system.

APPENDICES

A. Module Implementor Commands

B. Activity Coordinator Commands

C. Library Linkable Routines

D. Example of Building an Application

Module Implementor Commands

The following commands are implemented in the module implementor except as noted. The requests are message based and as such placed in a buffer and written to a network socket. The formats of the commands listed are described with the command in capital letters. These are defined in the library include file, "lib.h" and are all of storage class integer. The lower case variables are user defined with the storage class as indicated. The implementor returns an OK or FAIL (storage class integer) message for each command, except as noted.

BLOCK

Since the request for a connection results in a software signal sent to the module, the state of the signal acceptance must be conveyed to the implementor and made available to interested modules. The module may actually be in one of three states (in UNIX): ignore, block or accept. To put the module in a blocked state for incoming signals, the toggle is to be set for ON. After the CONNECT occurs, the module needs to perform this same command with toggle OFF (UNBLOCK).

The proper syntax for this command is:

```
     BLOCK, role, id, toggle
int  BLOCK;
int  role;
long id;
int  toggle;
```

CREATE

The activity coordinator receives a request (described in Appendix B) and sends the command to the appropriate implementor. The implementor will start the execution of the module. Upon proper commencement, a network address for the new module will be returned to the coordinator. The proper format of the request is:

```
     CREATE, id, name
int CREATE;
long id;
char * name;
```

The variable "name" is the full UNIX pathname of the module to be executed.

CONNECT

Any object of the system may request to open a communication channel with a module. The requesting module is returned as "request refused," "request completed," or "request failed." A request failed is returned if the module id sent does not exist on that machine. A request completed signifies that the id represented a module on that node, and a signal was sent to the module. The state of the module for accepting the signal is truly unknown. To initiate the request, the "toggle" is ON. When finished, the requesting module must reconnect to the target module's implementor with the toggle set to OFF to allow other modules to connect to the module. If the module is not accepting connections or another module is currently connecting to the module, a "request refused" is returned. The proper syntax of the request is:

```
     CONNECT, id, role, toggle
int  CONNECT;
long id;
int  role;
int  toggle;
```

DESTROY

The implementor will kill the execution of the module, giving prior warning to the module to allow "clean-up." If the module does not complete execution within a given time, the implementor will kill the module without further warning. This warning time of the system, WARN_TIME set in the library include file lib.h is currently 30 seconds. This warning time may vary depending upon the system and might either be changed at installation time or modified to allow the user to specify warning time lengths.

The proper format of the request is:

```
     DESTROY, id
int  DESTROY;
long id;
```

MOD_DEAD

In the event of a module death, the watcher sends this information to the implementor which will clean up all information regarding this module in internal databases. No arguments need to be passed with command since the implementor is able to determine which watcher the information is coming from by the value of the socket. The format of the command is:

```
     MOD_DEAD
int MOD_DEAD;
```

RECONNECTION

At one time, it was thought that socket connections may disappear and a module would need another socket. Upon placing the request, the implementor would appropriate another network port for the module, store the information for its own needs, and inform the coordinator of this occurrence. The coordinator would then inform interested parties of the occurrence and pass the replacement network information. I have not found a need for this functionality, and it is not currently implemented in the implementor nor in the minimal activity coordinator. The proposed format of the command was:

```
     RECONN, id, role
int  RECONN;
long id;
int  role;
```

Activity Coordinator Commands

CREATE

The ACM would not know actual implementation issues but would be knowledgeable about certain interfacing issues. Thus, the ACM might make decisions regarding the actual station on which a module would be executed. Since the coordinator acts in its place, this sort of information (as well as activity ID's) is read into the coordinator. Therefore, upon receiving a request for creation, the coordinator knows to which implementor to communicate. The syntax of the command is:

```
     CREATE, role, activity
int CREATE, role, activity;
```

The coordinator connects to the desired implementor and requests the creation. Upon proper creation the implementor returns the address of the module. This address is recorded by the coordinator. The coordinator returns to the requesting module a success or failure.

LOOKUP

A module may request information about all objects currently registered in a particular activity. The syntax of the request is:

```
     LOOKUP, activity
int LOOKUP, activity;
```

The following information is returned for each object currently found in the stated activity:

Network address: machine and port number
ID
Role

REGISTER

If a module desires to enter a new activity, it requests to be registered in that activity. This would allow death notices to be received by the module and other objects in the activity to determine which objects are in the activity (see Lookup).

The syntax of the command is:

```
     REGISTER, new_activity, id;
int REGISTER, new_activity, id;
```

DEREGISTER

If a module desires to end interest in an activity, it requests to be deregistered from that activity. Therefore, any future queries about this activity will receive no information regarding this module (see Lookup). This command returns to the requestor a list of module role tags of modules that are currently registered in this activity. Further, the AC sends a DEREGISTER command and the role tag to the other modules in the activity so socket connections may be cleaned up by those modules. The module role tags contained in the above list and those modules that receive the DEREGISTER command are not registered in another activity in which the requesting module is participating.

The syntax of the command is:

```
     DEREGISTER, activity, id;
int DEREGISTER, activity, id;
```

The following information is returned for each object which is currently registered in the specified activity. The objects referenced are not in another activity in which this abject is currently referenced. Thus, this information may be used to close socket connections for an activity in which this object is no longer interacting.

ID,Role

MOD_DEAD_UNNAT

MOD_KILLED

In the event of either command, the coordinator connects to each module in the module's activity(ies), to inform the module of the module's death. The coordinator then requests the re-creation of the module, receiving its new address and updating its database.

These commands would originate from the watcher, and the proper syntax for each is:

```
     MOD_DEAD_UNNAT, id
int  MOD_DEAD_UNNAT;
long id;

     MOD_KILLED, id
int  MOD_KILLED;
long id;
```

MOD_DEAD_NAT

When a module does finish complete execution and fulfills its task, it will commit suicide. In this event, the watcher will inform the implementor of the death and then inform the coordinator of the death. The activity coordinator will inform all interested parties of the death, but will not restart the module. The syntax of the command is:

```
     MOD_DEAD_NAT, id
int  MOD_DEAD_NAT;
long id;
```

Linkable Library Routines

The routines available in the linkable library file for activity development fall into two categories, routines with no need for global information, and routines that use global variables.

Independent Routines

create(module_role, activity_role);

```
int module_role;
int activity_role
```

This routine sends to the activity coordinator the module role to be created in the context of "activity_role". This routine is only useful if the activity coordinator has a priori knowledge of the files to be executed to fulfill that role within an activity.

int get_sock(system_user_id,port);

```
int system_user_id;
int *port;
```

This routine returns a valid socket which may be used to connect to other sockets or for binding and accepting connections. There should be no need for the application code to directly call this routine. This routine is currently called, in user code, from connect_activity. Modifications of the connections' interaction routines would utilize this routine. If the system_user_id is non-zero, the smallest available port value greater than 5000 is used. Otherwise, for the

system reserved ports, a port value greater than 4000 and less than 5000 is used. This routine returns a valid socket value (positive integer) or -1 on error.

int connecto(socket,machine_name,port_value);

```
int  socket;
char *machine_name;
int  port_value;
```

This routine attempts to perform a network connection to the port number and machine name passed. This routine should not be needed directly from the user code if the correct handler routines are implemented. The routine connect_activity() (below) does maintain a need for such a call. The creation of a different set of connection interaction routines would utilize this call. If a null pointer is passed as machine_name, the name of the the machine for which this library file was compiled is used. This routine returns a -1 on error.

int sconnecto(machine_name,port_value);

```
char *machine_name;
int  port_value;
```

This routine calls the above two routines but is included in this category since it utilizes no global information in performing its function. This routine calls get_sock and connecto with the information passed. A null pointer may be passed as the machine_name. On error a -1 is returned. Following the

premises of an activity-based system, there is no need for user implemented code to call this routine directly.

int sendtoAC(buff);
char buff[BUFFERSIZE];

Send the buffer to the activity coordinator. The return value is the return value from the coodinator from the request or -1 on local or communication error.

send_block(id);
long id;

This routine pends until it receives permission for the module to block. On communication or local error the routine exits.

send_unblock(id);
long id;

This routine pends until it receives permission for the module to block. On communication or local error the routine exits.

Routines Using Global Information

connect_activity(activity_role,module_role,id);

```
int  activity_role;
int  module_role;
long id;
```

This performs a LOOKUP by connecting to the coordinator.For each module received from the coordinator, the routine blocks itself (send_block) connects to the target module, completes the connection, and unblocks itself (send_unblock). This routine modifies, if there is no previous connection, the array **socket**. This is an integer array of size MAXROLES, set in the include files. The module role is the offset into an array that contains the communication socket value of the role of the module with which a module desires to communicate.

connect_handler();

This is the connection handler for the software signal received by the module from the implementor when the implementor has received an acceptable connection request. This handler is set up in the initialization routine (see "initial" below). If connect_activity is modified, connect_handler may need to be modified to reflect this modification.

int picksock(seconds);

int seconds;

This routine creates a mask from the valid open sockets contained in the "socket" array. A "select" is then performed with this mask with a time-out of "seconds" number of seconds. If no socket currently has a read condition outstanding, a zero is returned. This is the value of the standard socket and thus not a valid socket for a read condition. The value of the socket which has a read condition outstanding is returned. A value of zero seconds causes the routine to pend until a socket has a read pending on a valid socket.

initial(activity, module_role, id);

```
char *activity;
int  module_role;
long id;
```

This routine sets up the requisite handlers for the connection and calls connect_activity with the activity number of its initial activity. The actvity role, tag, is passed as the first argument to the command line. The module system id is passed as the second argument on the command line. The module role is user definable.

register(activity_role, module_role, id);

```
int  activity_role;
int  module_role;
long id;
```

This routine first sends a REGISTER request to the activity coordinator. Upon successful return, this routine calls connect_activity. Therefore, once the module is registered in a new activity, it is also connected to requisite modules before returning to the user code.

deregister(activity_role, id);

```
int  activity_role;
long id;
```

This routine first sends a DEREGISTER request to the activity coordinator (see Appendix B). Using the return information, the apropriate network sockets in the socket array may be verified and closed.

Example of Building an Application

For the purpose of describing the current environment and the capabilities of the final system, mostly the characteristics of the activity coordinator, and for the possibility of future development under this system, a description of the process of creating a new application is appropriate. The first section below describes the application objects that were introduced in Section 3.3, the nursery for weaned pigs on a hog farm. Following this is the a priori information required to execute under this developed system and steps to follow.

Application Overview

In the pig nursery, assume there are six pens for holding up to forty pigs each and the following objects:

3 fans	1 clock/timer unit
2 heating units	1 temperature monitor
3 windows	1 air quality monitor
1 main grain pipe	2 high pressure water pipes
6 water spickets	6 grain bins
6 water troughs	

The fans, heater units, and windows are evenly distributed throughout the available space. The air quality and temperature monitors are distributed in the building and are able to calculate an average of the quantity measured, or measure specific areas. The grain bins are fed off the main grain pipe with a flow valve to control the flow. Further, each bin has a scale within so the existing amount of grain

may be determined. Each spicket feeds into a water trough which also may contain a facility to measure current capacity. The floors of the pens are metal grates so that excreta will pass to a cement sub-flooring. The high pressure water pipes are under the grates and above the cement floor to flush the excreta.

The temperature and air quality monitors would continually measure those elements of interest. If the element is outside a given range, the monitor may request a fan or heating unit be created/registered in this activity. Further, the water in the troughs may be measured periodically, and the water may be turned on by the timer or by another manager requesting the object to be created at a given interval. The feeding would occur once per twenty-four hour period and would involve interaction of the main feeder line, the individual bins, the scale of each bin, and the flow control valve for each bin. The flushing of the excreta may occur four or more times a day, possibly dependent upon the total number of pigs in the house. These last two topics, food and flushing, also give rise to need for the system to know the number of pigs currently in the house. Therefore, there is the possibility of interaction with an overall farm inventory activity.

The following is one possible design of activity and object interactions. There is a need for at least five activities: environment control, feeding, watering, refuse disposal and farm inventory. Each activity will have need of time; therefore, the clock/timer object would interact in all activities and may be used to start the system.

The clock/timer object may interact in two possible fashions. In one mode, the clock is able to automatically request the creation of activities that need to occur every increment of time. In this situation, the activities may

complete the task at hand and commit suicide. Also, the clock/timer is able to queue up requests for wake-up calls. In this case, the activities might continually exist. One object in the activity requests to receive a message at a certain time or in a duration of time. Upon receiving the message, the object may determine the needs of the activity and continue execution. This design utilizes the former mode of operation.

Upon creation, the clock/timer will register in all activities and request the creation of the air quality monitor and temperature monitor in the context of the environment control activity. Then, it will request the creation of the six water spickets in the context of the watering activity. Dependent upon the time, it will then request the creation of the main feed pipe in the context of the feeding activity, and/or request the creation of the high pressure water pipes. At this point, we may now look at the interactions in each activity.

Farm Inventory: I will not discusstthis fullccapabilit and design beyond the needs of this environment. The activity is able to return the number and ages of the pigs in particular buildings on the farm. Various activities of the nursery request this information as discussed below.

Feeding Activity: The main feed pipe will register itself in the inventory activity to determine the number of pigs in each pen and thus grain needs. Upon completion of this request, it may deregister itself from this activity. The main feed pipe will then request the creation of needed grain bins. The grain bins may inform the main feed pipe of the current amount of feed or the need for grain, depending upon its intelligence. After sufficient grain is contained in a

bin, the bin will commit suicide. After all bins have died, the main feed pipe will also commit suicide. The clock/timer will recreate the activity at the required time.

Watering Activity: The individual water spickets will behave in a similar fashion as the grain bins with one addition: flushing of old water. The water spickets may, regardless of current capacity, flush water into the troughs for the purpose of flushing old stagnant water or refilling and flushing. After this goal has been accomplished, they will die off.

Refuse Disposal: The refuse disposal activity would consist of a single manager of the two high-pressure water pipes. The manager will register in inventory activity to determine the number of pigs in all the pens and the pens used since the last flush. The manager is then able to operate the pipes in such a fashion as to perform the cleaning. This activity may be created four or more times per day, depending upon actual need.

Environment Control Activity: The temperature monitor continually monitors air temperature. If the average temperature deviates from a given range, it may request the creation of a fan or heater. Further fans or heaters may be created from that fan or heater, but it would make more sense to have the monitor make further requests. Since the creations occur from the monitor, it would insure that hot air is not simply sent out via the fans.

The air quality monitor will work in similar fashion, except it may have a need to flush hot air. When there is a need to flush hot air, the air quality monitor will perform this function and inform the air temperature not to kill the

fans. When the air quality monitor is finished, it will rescind this order so the air temperature monitor will be able to kill them if necessary.

When any of the objects in the application receives a kill notice, it will immediately stop all operation and die off. This is in the case of emergencies and/or danger to the life of pigs and humans.

Requirements of the System

The current activity coordinator needs to have a file named "addresses" available to it upon execution. Each line in the file should contain for each module the following information:

```
"name          id       machine          role"

char  name[];
long  id;
char  machine[];
int   role;
```

The name is the UNIX pathname of the file to be executed to act as the object of interest, and this must conform to the file system capabilities and characteristics as defined by UNIX (i.e. remote file accessibility). The id is any value deemed suitable but must be unique for each object. The machine is the machine name of the node within the system on which the file is to be executed. The role is the role of the object as defined in Section 3.2. This value need not be unique system-wide but should be unique within an activity with careful consideration given to object roles in two or

more activities and the capabilities of the connection handler as previously described.

The module implementor needs no a priori information. All information and data bases are dynamically created and removed as the system grows and shrinks.

The network of computer nodes needs to be connected via standard TCP/IP, and information relating to network addresses and machine names need to exist as defined in the installation and configuration manuals.

To migrate the system to other systems, the machine names of the coordinator and the machine name of the target system for the linkable library file must be defined within parameter files. Both program files include the file "lib.h." The name of the activity coordinator machine is defined by "AC_MACH," and the target machine for the linkable library file is "THISMACHINE."

If the existing routines for connecting modules are going to be used, the value of the maximum number of roles and maximum number of socket connections should be checked. These values are defined by "MAXROLES" and "MAXSOCKS," correspondingly. For sanity's sake, these values should be the same but they do not have to be.

Finally, the path of the final executable file for the watcher on each system needs to be set. The implementor and watcher also use the include file "lib.h." The pathname of the watcher is defined by "WATCHER_FILENAME."

[BAN85] Banino, J.S. et al. "Some Fault-Tolerant Aspects of the CHORUS Distributed System" IEEE 1985 Distributed Computing Systems p 430-437

[ELL85] Ellis, Carla and Heliotis, James "Structuring Resilient Distributed Programs with the Activity Model" U of Rochester TR #163

[FEL79] Feldman, Jerome "High Level Programming of Distributed Computing" Communications of the ACM June 1979 22,6 p 353-367

[FIS85] Fischer, Michael J. et al. "Impossibility of Distributed Consensus with One Fault Process" Journal of the ACM April 1985 32,2 p 374-382

[GAI85] Gait, J. "A Distributed Manager with Transparent Continuation" IEEE 1985 Distributed Computing Systems p 422-429

[HEL84] Heliotis, James "Language Constructs for the Management of Distributed Computations" Doctoral Thesis from the University of Rochester

[HRE81] Hrechanyk, Lydia "Process Management on VAX/UNIX" May 1981 Personal paper

[JOH84] Johnson, Dave "The Intel 432: A VLSI Architecture for Fault-Tolerant Computing Systems" Computing Aug. 1984 p 40-48

[JOS86] Joseph, Thomas A. and Birman, Kenneth P. "Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems" ACM Transactions on Computing Systems 4,1 p 54-70

[JUL87] Jul, Eric et al. "Fine-Grained Mobility in the Emerald System" Operating System Review Proceedings of the Eleventh ACM Symposium on Operating System Principles Nov. 1987 p 105-106

[LAM82] Lamport, Leslie et al. "The Byzantine General Problem" ACM Transactions on Programming Languages and Systems July 1982 4,3 p 382-401

[LAM85,Jan] Lamport,Leslie and Melliar-Smith, P.M. "Synchronization in the Presence of Faults" Journal of the ACM Jan. 1985 32,1 p 52-78

[LAM85,Feb] Lamport, Leslie and Chandy, K. Mani "Distributed Snapshots: Determining Global States of Distributed Systems" ACM Transactions on Computing Systems Feb. 1985 3,1 p 63-75

[LAP85] Laprie, J.C. "Dependable Computing and Fault Tolerance: Concepts and Terminology" IEEE 1985 Fault Computing Conference p 2-11

[LIS87] Liskov, Barbara "Implementation of Argus" Operating System Review Proceedings of the Eleventh ACM Symposium on Operating System Principles Nov. 1987 p 111-122

[NEG84] Negrini, Roberto et al. "Fault Tolerance Techniques for Array Structures Used in Supercomputing" Computer Feb. 1984 p 24-45

[SAR84] Sarrzin, David B. and Malek, Miroslaw "Fault-Tolerant Semiconductor Memories" Computing Aug. 1984 p 49-56

[SCH83] Schlichting, Richard D. and Schneider, Fred B. "Fault-Stop Processors: An Approach to Designing Fault-Tolerant Systems" ACM Transactions on Computer Systems Aug. 1983 1,3 p 222-238

[SCH84] Schneider, Fred B. "Byzantine Generals in Action: Implementing Fail-Stop Processors" ACM Transactions on Computer Systems May 1984 2,2 p 145-154

[SER84] Serlin, Omri "Fault-Tolerant Systems in Commercial Applications" Computer Aug. 1984 p 19-30

[SIE84] Siewiorek, Daniel P. "Architecture of Fault-Tolerant Computers" Computer Aug. 1984 p 9-18

[SKE85] Skeen, Dale "Determining the Last Process to Fail" ACM Transactions on Computer Systems Feb. 1985 3,1 p 15-30

[SMY86] Smyth, David Private communications on UUCP April 1986

[STR85] Strom, Robert E. and Yemini, Shaula "Optimistic Recovery in Distributed Systems" ACM Transactions on Computer Systems Aug. 1985 3,3 p 204-226

[SUN85] SUN Microsystems Inc. "UNIX Interface Reference Manual" signal(3) p 178-180

[TOY84] Toy, Wing N. and Morganti, Michele "Fault-Tolerant Computing" Computer Aug. 1984 p 6-8

[UNI83] Leffler, Samuel et al. "UNIX Operating System - 4.2BSD Interprocess Communication" July 1983

[WAL84] Wallace, John J. and Barnes, Walter W. "Designing for Ultrahigh Availability: The UNIX RTR Operating System" Aug. 1984 p 31-39

[XER81] "Courier: The Remote Procedure Call" Xerox Corporation, Stamford,Conn.

[YAN85] Yang, Y.Z. et al. "Fault Recovery of Triplicated Software on Intel APX 432" IEEE 1985 Distributed Computing Systems p 438-445