Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

1988

Implementation of an activity coordinator for an activity-based distributed system

Robert Shaw

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Shaw, Robert, "Implementation of an activity coordinator for an activity-based distributed system" (1988). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Rochester Institute of Technology
School of Computer Science and Technology

Implementation of an Activity Coordinator for an Activity-Based Distributed System

Robert Shaw

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

Approved by:

Dr. James E. Heliotis 11/16/88

Dr. Andrew T. Kitchen 11/16/88

Dr. Peter G. Anderson 15 Nov 88

I,Robert Shaw, do hereby grant permission to Wallace Memorial Library of R.I.T., to reproduce my thesis in whole or in part. Any reproduction will not be used for commercial use or profit.

12/12/88

Implementation of an Activity Coordinator for an Activity-Based Distributed System

Robert Shaw

Abstract

Distributed computing systems offer a number of potential benefits, including:

- improved fault-tolerance and reliability
- increased processor availability
- faster response time
- flexibility of system configuration
- effective management of geographically distributed resources
- integration of special purpose machines into applications

In order to realize this potential, support systems that aid in the development of distributed programs are needed. An *Activity System* facilitates the design and implementation of distributed programs:

(1) By allowing the programmer to group functionally related objects into an activity (or job) which is recorded within the system. The information stored concerning relationships between objects may then be used to control their interactions and thus to manage distributed resources.

(2) By effectively eliminating the need for the programmer to deal with the underlying details of inter-process communication. The system handles the establishment of communication links between objects in an activity, and controls the routing of messages to activity members.

To evaluate the uses of activities in developing distributed programs, I have implemented a portion of such a system; namely, an *Activity Coordinator* , together with Activity System components and test tools required to verify its functionality. Within the context of an Activity System, the Activity Coordinator provides certain key functions:

(1) It maintains a database of information pertaining to objects and activities, and

(2) It handles the routing of activity related messages.

In future versions of the activity system the Activity Coordinator may also play a more active role in fault recovery. These possibilities will also be discussed.

Table of Contents

Introduction 1

Chapter 1. Distributed Systems 3

1.1 Objectives 3

1.2 Issues 4

1.3 Definitions 5

Chapter 2. Traditional Methods 6

2.1 IPC Classification 6

2.2 RPC - Courier 7

2.3 Rendezvous - XMS 8

2.4 Port-Based - Accent 9

Chapter 3. Recent Systems 10

3.1 The Object Model 12

3.2 Transactions 16

3.3 Objects and Actions 18

3.4 The Role of Processes 21

3.5 Active vs Passive Objects 22

3.6 Cronus 23

3.7 Argus 25

3.8 Clouds 28

3.9 Archons 31

3.10 TABS 33

3.11 Real-time Process Control 36

3.12 CONIC 38

apter 4. Activity System 41

4.1 System Description 41

4.2 A Sample Activity 46

4.3 Activities vs. other systems 47

apter 5. Implementation 49

5.1 AC Functions 49

5.2 AC Design 52

apter 6. A Simulation 57

6.1 System description 57

6.2 Simulation Design and Implementation ... 59

6.3 Object Implementor 59

6.4 Name Server 63

6.5 Simulation - Objects and ACM's 65

6.6 System Startup and Operation 69

6.7 Test Tool Programs 72

apter 7. Implementation Issues 73

7.1 Database Issues 73

7.2 Communication Issues 79

7.3 Simulation Issues 82

apter 8. Conclusions, etc. 86

8.1 Conclusions 86

8.2 Issues and Related Topics 87

8.3 System Performance 89

8.4 Future Work 91

Introduction

Distributed computing systems offer a number of potential benefits, including:

- improved fault-tolerance and reliability
- increased processor availability
- faster response time
- flexibility of system configuration
- effective management of geographically distributed resources
- integration of special purpose machines into applications

In order to take full advantage of this potential, support systems that aid in the development of distributed programs are needed. The *Activity Model* [Heliotis84] provides a number of useful tools for this purpose. To evaluate the uses of activities in developing distributed programs, implementation of at least a prototype activity system is required. To this end, I have implemented a portion of such a system; namely, an *Activity Coordinator* , together with Activity System components and test tools for verifying its functionality. The purposes of an activity system and the role of the activity coordinator are outlined below.

An activity system facilitates the design and implementation of distributed programs:

(1) By allowing the programmer to group functionally related objects into an activity (or job) and to record this relationship within the system. This information concerning relationships between objects may then be used to control their interactions.

An activity may be further divided into sub-activities, thereby permitting a logical hierarchy to be built. The system also provides mechanisms for managing the dynamics of the activity structure thus created.

(2) By effectively eliminating the need for the programmer to deal with the underlying details of inter-process communication. The system handles the establishment of communication links between objects in an activity, and controls the routing of messages to activity members.

As we shall see, the activity model allows programmers to construct reliable distributed programs while retaining "flexibility of management options" [Ellis85].

Within the context of this system, the Activity Coordinator (AC) provides key functions: (1) It maintains a database of information pertaining to objects and activities, and (2) It handles the routing of activity related messages. In future versions of the activity system the AC may also play a more active role in fault recovery.

Chapter 1 outlines the objectives of distributed computation and discusses the issues related to achieving the potential benefits. In Chapter 2, I will discuss 'traditional' communications-oriented approaches to distributed computing. Chapter 3 describes more recent work in language/system support for distributed programming, including a discussion of the object model and atomic actions. Chapter 4 describes the activity model in detail, focusing on characteristics which distinguish it from other recently proposed systems. Chapter 5 details my design and implementation plan for the Activity Coordinator. Chapter 6 outlines the test procedure for the coordinator and describes a system built for this purpose. Chapter 7 addresses issues involved in the actual implementation of the coordinator. Pertinent topics in inter-process communication and distributed database management will be discussed. Finally, Chapter 8 presents conclusions and suggestions for future work in the development of an activity system.

Chapter One

1.1 Objectives of Distributed Processing

The objectives of distributed computing can be described in terms of the benefits that can be achieved in the following areas:

1. Reliability - refers to the period during which system services are provided without interruption, and is usually expressed in terms of Mean Time Between Failures (MTBF) and Mean Time To Repair (MTTR). The fact that more processors are in use in a distributed system increases the probability that at least one of them will be functioning at any given time. The system may thus be able to survive the failure of individual components while retaining a high degree of functionality.

2. Availability - Closely related to reliability is availability. Availability is the fraction of time that system components are operational, and can be computed as MTBF/(MTBF+MTTR). Availability is dependent on the level of redundancy and ability to reconfigure the system. By having the ability to put replicated resources to work, the overall availability of the system can be greatly increased.

3. Performance - Better performance can be achieved in a number of ways:

a) Faster response time can be achieved by exploiting parallelism inherent in applications.

b) Overall performance can be improved by balancing the load on system resources.

c) In some cases, integration of special purpose machines can result in greater efficiency. Instead of using general purpose machines for executing a variety of independent tasks, each processing element can be specialized to perform the required tasks as efficiently as possible.

4. Resource sharing - A distributed system architecture permits effective use of physically dispersed and/or expensive resources by multiple users.

5. Extensibility - Distributing hardware and software components allows for greater flexibility of system confi-

guration. The modular nature of a distributed architecture permits processing elements to be removed or added easily.

1.2 Issues in Distributed Processing - Achieving the Objectives

The benefits of distributed computing do not accrue automatically from simply connecting together hardware components. In order to achieve the objectives outlined above, new issues must be addressed:

1. Reliability - to achieve improved reliability in spite of processing failures, software must be provided to perform error detection, diagnosis, and recovery (including re-configuration, if necessary).

2. Availability - the use of more processors increases the probability that any one of them will fail over a given period of time. In addition to making each component highly reliable, a high degree of redundancy can improve the overall system availability. Additional software is required to manage the use of replicated resources.

3. Performance - to achieve increased performance, resources must be distributed in such a way that the following criteria are met:

a) Computation is performed in such a way that concurrent processing can occur, and thus, unnecessary sequentialization should be avoided. For general distributed computing, determining possible parallel executions is a non-trivial problem. Furthermore, system control is needed to detect and resolve conflicting processing requests without impairing parallelism.

b) The load on resources (processors, memory, communications) is balanced (at the very least such that none is overloaded, and ideally such that each component achieves optimal performance).

c) The overhead incurred by communication does not override the benefits obtained through distribution over many (possibly specialized) machines.

4. Resource sharing -

A support system for distributed computing should provide system-wide control of activities for the purpose of achieving optimal resource utilization. Techniques used in a central processor for controlling access to shared resources typically involve serializing the execution of potentially competing parallel processes. In a distributed system, a synchronization mechanism that preserves concurrency is required.

5. Extensibility - It should be possible to reconfigure processing elements dynamically without disrupting the system. Configuration control must be provided that minimizes the impact of such changes on the system.

1.3 Distributed Systems - Definitions

The hardware components of a distributed system may be viewed as a set of *nodes* connected via a communications network, where a node consists of processor(s), memory, and any number of external devices. A *distributed program* is a set of modules that run on nodes and which cooperate (by exchanging messages) in order to achieve their goals. Physically distributed hardware and a collection of distributed programs do not, however, constitute a distributed system. In addition there must be some type of system support for handling interprocess communication, synchronization, and recovery.

Before discussing these issues it is important to note certain characteristics of a distributed system which affect the possible solutions: There is no sharing of memory among process modules, and all communication is accomplished via links between nodes. Interprocess communication delays are variable and non-zero. As a result, some time always exists between the production of an event and the realization of that event at its destination. This is in addition to the delay that exists for the event to be observed by other modules. Whereas in shared memory multiprocessors the exchange of information between modules occurs in microseconds, in a distributed system it may take many milliseconds. Due to the distribution of state information and message induced time delay, it is not possible for a single module to have a complete current view of the entire system state. As a result, system control is performed by several modules which cooperate to provide synchronization, recovery and runtime management. It is this distributed computation management - in particular, within the framework of the activity model - that is the focus of this thesis.

Chapter Two

Traditional Methods

The traditional approach to designing distributed systems has emphasized communications, with the goal of making networking and distribution of resources transparent to the user. As a result, these systems have concentrated on the structure and semantics of inter-process communication (IPC).

2.1 IPC Classification

One way of classifying message passing paradigms is by their communication structures:

(1) one-to-one - occurs between two specific processes. Specifying the processes in communications statements creates a static communication channel by *direct naming*. This method of communication is employed by CSP [Hoare78].

(2) one-to-many - a process may need to *broadcast* a message to multiple destinations. This is frequently the case in real-time systems where the output from one sensor may be required by several controllers and monitoring devices. Other examples include distributed applications in which information from one node is shared with other nodes in the system – such as distributed routing algorithms, automated load balancing, and name servers.

(3) many-to-one - several *client* processes may communicate with one *server* process, as for example when several users share a single device. The client/server model is the most commonly used approach in constructing distributed applications. Such systems normally operate as follows: A server process listens at a well known address for service requests. Client processes request services by initiating a connection with the server. In a distributed environment there must also be a mechanism whereby clients can access remote servers. This typically involves the use of name servers to locate the target for requests. In developing client and server applications, the protocol for making and accepting service requests, as well as for remote service access, must be established beforehand and implemented by both ends of the connection. Of course, variations of this protocol exist depending on the semantics of communication (see below - Courier, XMS).

(4) many-to-many - arises when there are several clients requesting services from any one of a number of identical servers.

IPC mechanisms may also be classified based on the synchronization properties of message passing. In asynchronous message passing, the execution of a send statement does not delay the sending process. In order to achieve this non-blocking send, however, there must be buffering of messages between sender and receiver. If buffering is not available, the sender is delayed until the message is received. This is known as synchronous message passing. In Remote Invocation Send, the sending process waits not only for the message to be received, but also for a reply to be returned. On the receiver side, message receipt may either be explicit using a blocking receive statement, or implicit (non-blocking) by invoking some code module similar to an interrupt handler.

The following sections serve to illustrate how these communication mechanisms are used in certain representative implementations.

2.2 Remote Procedure Call - Courier

Remote invocation send together with implicit receive constitutes Remote Procedure Call (RPC). This is the mechanism used by the Courier system [Xerox81]. In this system, a single passive listening process - the Courier server - resides at a well known address on each machine. Client processes (typically application programs) comprise the active system elements. When a client process initiates a connection to the Courier socket, the listener spawns a server to attend to the user request. Courier handles the underlying communication calls between machines so that a remote procedure call behaves from the user perspective as if the procedure were performed locally. Unfortunately, the synchronous message passing semantics of RPC limits parallelism. First, since Courier is based on virtual circuits the caller must await a connection with the server. Second, remote procedure calls must execute sequentially, not concurrently, since the caller must await results.

2.3 Rendezvous - XMS

Combining remote invocation send and explicit receive statements is known as Rendezvous [Ada83]. The event sequence for rendezvous is shown in figure [2.1]. A rendezvous occurs as follows: Task A invokes an entry declared in the interface specification of task B. Task B executes an accept statement enabling a call to that entry. Input parameters are passed from the invoker to the accepter at the start of the rendezvous, and output parameters are passed back to the invoker when the rendezvous ends. Communication in the XMS system [Gammage85] is based on an extension of this local rendezvous mechanism for distributed systems, namely *remote rendezvous*. The semantics of remote rendezvous are nearly identical to that of local rendezvous, except that parameters are transmitted as data packets over a local area network, and there is additional communication required in the form of acknowledgements. Figure [2.2] shows the protocol for remote rendezvous.

To invoke a remote rendezvous a task must know the interface of the remote task it wishes to communicate with and a task id for it. Remote task id's may be obtained from a name server on the local node. A task (server) that can be invoked remotely registers itself on its node. All name server tasks communicate to exchange names and task id's of registered tasks, so that any registered task can be invoked from any node in the system. This arrangement promotes a client/server relationship between tasks. Accordingly, "the preferred approach for XMS is to use a client-server model of the application" [Gammage85]

Like RPC, rendezvous is a synchronous message passing scheme, and thus it suffers from the same limitations with respect to concurrency. Note that regardless of which task (invoker or accepter) begins the rendezvous there are two wait states involved: prior to the copy of input parameters to the accepter's address space and prior to the copy of output to the invoker's address space (see figure [2.1]). If an a application requires non-blocking interactions among tasks, creation of a separate communications subtask is required. The subtask performs the interaction on behalf of its parent. However, it must then rendezvous with the parent in order to return results. Thus, concurrency gained at one point in the computation is lost during the ensuing rendezvous.

2.4 Port-based Communication - ACCENT

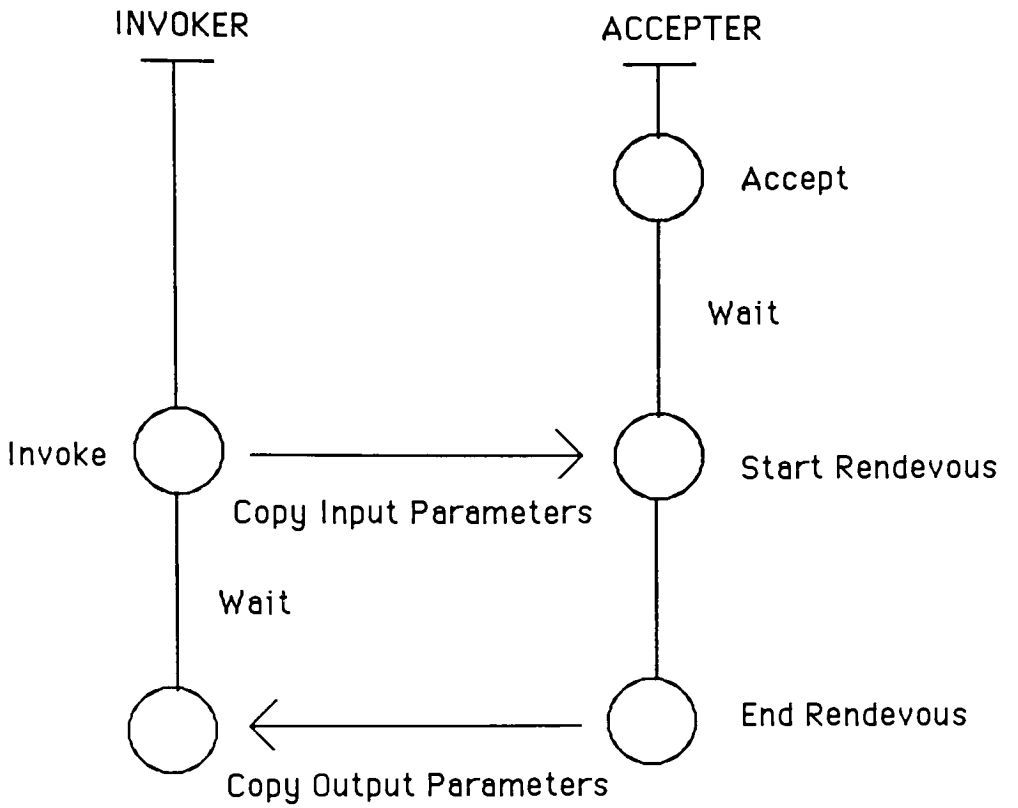
A port-based communication scheme is designed to allow asynchronous message passing and a many-to-one structure. Ports are the foundation of the CMU IPC facility, provided as part of the ACCENT network OS kernel [Rashid81]. In the ACCENT system, a port is a FIFO queue of messages contained in the kernel. Any process that can refer to a port by name may place messages into the queue. A process may remove messages from a port provided it has *receive access* to that port. In the ACCENT system, only one process has receive access to a port at a time, although receive rights can be transferred between processes. This allows a process to take over services provided by another process should that process fail. With a slight modification to this paradigm ports can be used establish a many-to-many structure. This is accomplished by allowing multiple processes to have receive rights, and providing a mechanism for viewing messages without removing them from the queue.

The problem with such message based systems occurs in the handling of failures and in process synchronization. Since there is no inherent structure in how messages are passed, interdependencies between processes can not easily be determined. Unless explicit measures are taken to ensure progress in computations (e.g. incorporating time-outs in applications), message-based deadlock is possible. If a process or site fails, the effects on cooperating processes and sites is often not clear. This makes error recovery more difficult. Each process must determine a course of action based only on local information, "even when a global context may allow a better recovery strategy" [LeBlanc85]. What is lacking is a method for defining and managing system-wide interactions between the various elements involved in a distributed computation. More recent research efforts in distributed computing have focused on precisely this problem.

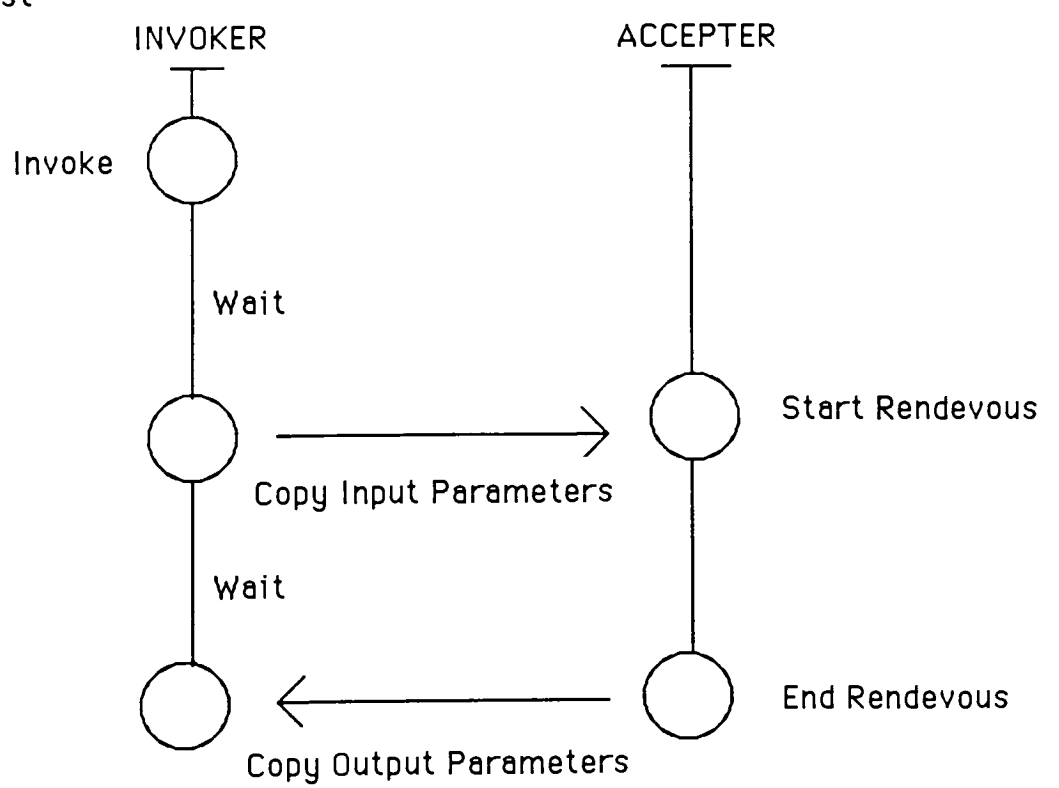
Having outlined some basics of IPC facilities, it should be noted again that the addition of communication mechanisms to physically distributed hardware does *not* make a distributed processing system. Rather, IPC facilities provide the foundation upon which distributed systems management is built.

Figure 2.1 - Rendevous Event Sequence

(A) Accepter First



(B) Invoker First



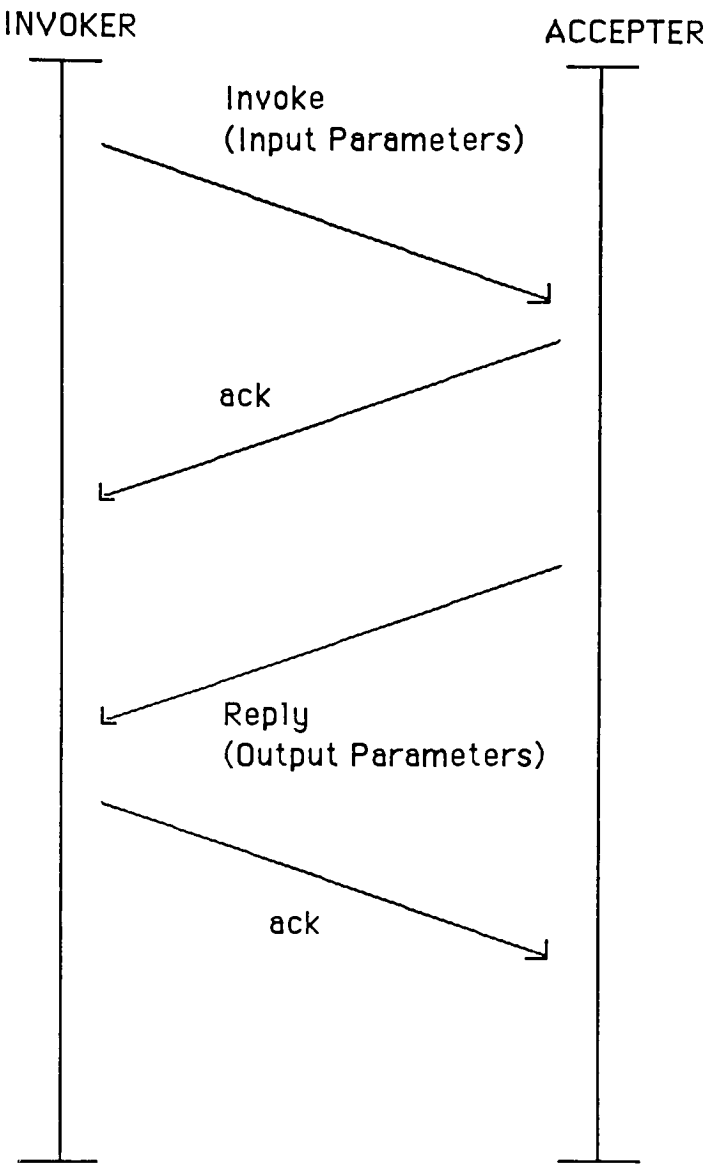


Figure 2.2 - Remote Rendevous Protocol

Chapter Three

Recent Systems

Recent related work in distributed computing support has focused on the following areas:

1) Database management,

2) Operating Systems,

3) Real-time process control,

4) Programming Languages

This paper will concentrate on operating system and real-time projects. Database systems are primarily of interest because of their use of the *transactions* – the precursor to *atomic actions* in operating systems. The transaction model will be examined in this context in a later section. However, a complete discussion of the recent developments in Database Management Systems is beyond the scope of this thesis. Similarly, programming languages will be considered only to the extent that certain developers have integrated a particular language into their systems. Details of these and other languages for distributed systems will be not be discussed here.

Although developments in databases and distributed operating systems "seem to be merging" [Moss85], there are nevertheless fundamental differences in the reliability (synchronization and recovery) requirements for each type of system and thus in the techniques used for achieving these requirements. These differences arise from the nature of the basic objects and their operations in each type of system. The section entitled "Operating Systems: Objects and Actions" will treat this subject in detail. In so doing, we will see how traditional operating system and database techniques must be modified to meet the needs of distributed systems.

The use of object/transaction techniques in real-time applications necessitates some further modifications. In considering distributed process control, synchronization of actions becomes an increasingly important factor. In such systems, partial or even total ordering of events is often not sufficient. Instead, real-time synchronization of operations is required. This requirement imposes additional performance constraints on the support system. It also implies the establishment of some mechanism for global clock synchronization. Discussions of real-time systems in this paper

will assume (whenever necessary) that such a mechanism exists. Clock synchronization is discussed elsewhere in [Lamport78], [Lamport82] and [Marzullo85].

Before describing particular research projects presently under development, I will examine the two models that are prevalent in current distributed systems research: *objects* and *transactions*.

3.1 The Object Model

Objects can be viewed as abstractions of system components. These components possess certain characteristic, *invariant properties* that determine their behavior. An object can only be manipulated by a set of *operations* that preserve these properties. Thus, in effect, the behavior of an object is defined by its operations. A queue object, for example, might well be represented as follows: It contains the actual queue as data. It has operations "enqueue" to add an item, and "dequeue" to remove an item. In addition, it possesses the invariant property that no more items can be removed from the queue than have been added to it.

The principal reasons for adopting the object model are modularity, simplicity, protection, synchronization and recovery. Much research has been done in the first three aspects of object-based programming. More recently, mainly in the context of developing distributed operating systems, researchers have begun to look at the advantages provided by the object-oriented approach in the areas of concurrency control and recovery.

The object model provides *modularity* by encapsulating data and operations in a single entity. Details of implementation, both of data structures and of the algorithms used by operations, are hidden behind the object boundary.

The object model promotes *simplicity* , especially at the level of system development. Object types and operations can be used without regard to data representations and implementation details. At the same time, objects correspond to real world entities whose behavior is reasonably well understood. Thus, system design becomes primarily a mapping process between the goals of the system and the appropriate objects. Object-based programming also makes definition of new modules simpler by virtue of *inheritance*. New object types can be defined simply by specifying how they differ from existing, more generic types, without having to 'start from scratch' each time.

Protection facilities are provided by an operating system to constrain the way information can be used and modified. A simple and straightforward mechanism for controlling manipulation of objects is to place restrictions on access to an object's operations. *Rights* or *capabilities* to perform operations are granted only to those who should be able to access that object.

The object concept represents a powerful *recovery* tool. Each object can be programmed to handle failures, and the encapsulation of objects provides for failure containment.

Lastly, *synchronization* facilities are also affected by the adoption of the object model paradigm. Under such a system, synchronization constraints are expressed in terms of permissible operation sequences on objects. This notion forms the foundation for *path expressions* (described below). In actual implementation, synchronization may be provided statically by the language system or dynamically by the operating system. The following is an overview of commonly used methods for implementing synchronization of access to objects:

Semaphores can be used in solving general synchronization problems, involving protection of resources inside *critical regions*. The mechanism requires that processes share access to a common semaphore variable, say "s". Each process must execute an atomic operation P(s) before entering the critical section. A process is blocked if the current resource count associated with s is 0. Otherwise, the process proceeds and the resource count associated with s is decremented. Each process must execute an atomic operation V(s) before leaving the critical section. This increments the resource count associated with s and allows blocked processes to enter. If this sequence is not followed, synchronization errors may result. Note that this problem will arise if even a single process does not follow the correct procedure.

Like semaphores, *conditional critical regions* can be used in solving general synchronization problems. Conditional critical regions take the form *region* v *when* B *do* S; where B is a boolean expression, v the shared variable, and S the statement(s) to be executed. When a process enters the critical region, the expression B is evaluated. If B is true, S is executed; otherwise the process is delayed until B becomes true *and* no other process is in the region associated with v. The main problem with conditional critical regions is that the conditions "B" must be evaluated very frequently and within the context of the caller, unless restrictions are put on B. This evaluation must occur:

- for deadlock avoidance, if a process leaves the region or is blocked by another wait clause,

- for minimal blocking, after every change of state variables [Lagally79].

Whereas both semaphores and conditional critical regions require each procedure to provide its own synchronization explicitly, *monitors* provide mutual exclusion synchronization automatically. The monitor construct contains both data and procedures needed for allocating its resources. Monitors ensure mutual exclusion by allowing exactly one process at a time to enter. Interaction among processes sharing the monitor is provided through *signal* and *wait* operations. The operation "wait(condition)" blocks the calling process and places it in a queue associated with

'condition'. The operation "signal(condition)" releases a process from the 'condition' queue and allows it to enter the monitor. The main problem occurs with nesting of monitor calls and concerns the semantics of locking. If monitors are nested, processes may be blocked unnecessarily; this is due to the fact that monitors automatically provide mutual exclusion on the data object, whereas mutual exclusion on the state variables would suffice.

A *path expression* has the form *path* S *end* , where S denotes a legal sequence of execution steps. This sequence is described according to the following syntax:

- S = S1;S2 means that the subsequence S1 must be followed by S2 and vice-versa.

- S = S1,S2 means that either S1 or S2 must follow.

- S = {S1} means that after S1 has been initiated, an arbitrary number of instances of S1 may occur. Only after all instances have completed may other steps proceed.

- S = (S1 - S2) means that the sequence S1 must occur at least as often as S2, but not more than "n" times more often.

Path expressions may be translated into an equivalent sequence of P and V operations and are thus logically equivalent to semaphores. One of the main drawbacks of path expressions is that the identity of the calling process is not a part of the description. If the state of the object depends on the relationships between processes accessing it, the formulation of the appropriate path expression becomes unclear [Lagally79].

The *object manager* construct is more recent than those previously described, and it fits most naturally with both distributed systems and the object model itself. In this approach, a process called an "object manager" is associated with each object. The manager controls access to the object in an application dependent manner, including handling synchronization. (In some of the systems we will discuss, object managers also play an important role in recovery).

Object managers represent an improvement over traditional operating system synchronization methods for a number of reasons [Lagally79]:

- Conditional critical regions and path expressions use only a localized view of the data objects. As a result, global considerations, including scheduling of operations and freedom from deadlocks, cannot be expressed in a

natural way.

- Nesting of Monitor calls or Critical Regions can lead to unwanted sequentialization.

- In a system that uses object managers, processes are loosely-coupled. This method is thus more appropriate for a distributed processing environment.

- In all the synchronization mechanisms mentioned, except object managers, access operations are called as procedures from the calling process. If a process is preempted while executing the operation, the object may be locked indefinitely. Use of object managers is the only method that provides a solution to this problem. Since actual invocation of object operations is handled by the object manager, the object is not kept locked if a user is preempted while accessing it.

While each object can be programmed to handle synchronization and recovery from a local perspective, objects do not work in isolation, but rather as components of larger systems. Thus, given an object-oriented programming environment, it is clearly desirable to have a mechanism for defining relationships between objects and for controlling their interactions. Transactions (or in operating system terms, 'actions') provide this mechanism.

3.2 Transactions

A transaction, in database systems, refers to a sequence of operations on database objects that, when finished, preserve their consistency constraints. Transactions possess the following properties:

- Failure atomicity (Totality): Either all or none of a transactions operations are performed. If a transaction does not complete, its partial results are undone so as to insure that it has no undesired side effects.

- View atomicity (Concurrency transparency): A transaction appears to take place indivisibly, without concurrent interference, and an incomplete transaction cannot reveal its results to other transactions. This prevents 'cascading aborts' in case the transaction must later be undone, and thus provides for failure containment.

These first two properties are often referred to together and termed simply "atomicity". However, it will be useful in future discussions to distinguish between them. That is, between undoing a transaction's unsuccessful operations and "hiding" partial results of operations from other transactions.

- Permanence: If a transaction completes successfully, the results of its operations will not subsequently be undone.

- Serializability: If several transactions execute concurrently, their effect on the database is the same as if they were executed serially according to some correct schedule.

Transactions are run concurrently, with their operations interleaved. An interleaved execution of transactions is known as a *schedule*. The *scheduler* or *transaction manager* is responsible for safeguarding the consistency of the database by producing only correct schedules. A schedule is correct if it satisfies the following conditions: (1) There exists a total ordering of the set of transactions, and (2) For every pair of conflicting operations (i.e. operations that access the same object), their relative order on the shared object is the same as their corresponding order in the total ordering of transactions [Papadimitriou86]. This theoretical result is can be realized with either locking or

timestamps. If locking is used, then the following test produces the desired result: A transaction step may proceed, unless it conflicts with a previous step of another active transaction. To carry out the test, the scheduler checks whether any incomplete transactions hold locks to the object that are incompatible with the current operation. If timestamps are used, the following mechanism may be applied: A step may proceed if the timestamp of its transaction is larger than the timestamp of the object accessed. If the step proceeds, the timestamp of the object accessed is updated to the timestamp of the transaction. In either case, the effect is the same: The scheduler may be required to delay the execution of a transaction in order to preserve serializability.

In Database Management Systems, the purpose of transaction manager or scheduler is to maintain consistency constraints on stored data, without unduly restricting concurrency. In operating systems, on the other hand, the goal is to maximize parallelism and resource utilization. As we shall see, it is possible in an operating system context to formulate non-serializable transactions (actions) that improve concurrency while maintaining consistency requirements. Thus, in order to extend transactions to an operating system environment several modifications to the model must be made.

3.3 Operating Systems: Objects and Actions

First we must consider the nature of objects in the operating system context, as opposed to that of a database system. In particular, we are concerned with their operations and recoverability properties. The basic objects in a database system consist of records collected into files. The only operations on records are read and write, and only operations such as insert, delete, lookup and sort (which are simply variations on read and write) are defined for files. [Spector] Moreover, database objects can be restored in the event of an action failure. Operating system objects typically represent system resources: physical entities such as disk drives or printers and data abstractions such as directories or queues. In order to provide for data abstraction, the system must support arbitrary object type definitions with corresponding type- specific operations, instead of simply read and write. Moreover, in operating systems, not all objects are recoverable. A resource is non-recoverable if its use changes its state irrevocably. Changes to such objects cannot be deferred until commit, nor can they be undone upon abort. Examples of such objects in operating systems include disk sectors, tape drives, printers and buffers [McKendry84]. Recoverability ultimately rests on level of abstraction of the object: The question is essentially one of abstract versus physical entities. For example, the value of a bank balance may be temporarily suspended between old and new versions, pending the completion of a transaction. If the transaction commits, the new version becomes the "real" value. If the transaction aborts, the old value may be restored. The use or non-use of a disk sector can not be similarly suspended. The effects of its use are immediate: Once written, the value automatically becomes the current version for that sector and the previous version is lost. Under these circumstances, recovery mechanisms used by database systems (i.e. undoing and redoing transactions) can not be applied.

Next, we must consider the difference between atomic actions and database transactions. In database systems, transactions use only syntactic and very limited semantic information (i.e. conflict knowledge) in order to achieve serializability. In operating systems, greater concurrency can be achieved by using additional semantic knowledge both about objects and about the properties of transactions in which they are involved. For example, in order to realize concurrency transparency, it is not always necessary that execution of atomic actions be serializable [McKendry84],[Spector], [Allchin83] The following examples serve to illustrate this point:

(1) Consider a queue that buffers units of work between producer and consumer transactions. Serializing the transactions that operate on the buffer requires that all entries made by a single transaction be removed in the order of entry. (In other words, consecutive removal of items is enforced). However, in many producer/consumer problems of this kind the queue need not be treated as strictly FIFO (i.e. a *weak queue* may be more appropriate). So long as entries for which the inserting transaction has committed are eventually removed from the queue, and items inserted by aborted transactions are not removed, atomicity is maintained [Spector].

(2) Suppose we have a storage map object S whose purpose is to allocate disk pages to the user. S has two operations: Get() and Put(). Get examines the current state of the map and returns a free page if one is available. Put returns a currently allocated page to the map. Consider the following sequence of events involving actions A1 and A2:

A1 requests a Get() and is allocated page p1

A2 requests a Get() and is allocated page p2

A1 requests a Get() and is allocated page p3

Ignoring the user view (abstract level) and considering only the implementation view (physical level), serializability requirements dictate that this sequence not be allowed, since it violates the relative ordering condition. The transaction A1 must be allowed to complete without concurrent interference from A2 and thus must be allocated consecutive pages. This problem arises because, from the implementation standpoint, not all pages are identical. From the standpoint of the user, however, this interleaved sequence may be considered acceptable, because it allocates the pages as requested. Thus, view atomicity is maintained at the abstract level despite non-serializability at the physical level [Allchin83].

The key difference between actions in an operating system and database transactions is the absence of the serializability requirement in the former. In discussing actions, therefore, we will be concerned with the two atomicity properties: concurrency transparency (view atomicity) and totality (failure atomicity).

View atomicity is linked to synchronization, while failure atomicity is the concern of the recovery mechanism. Allchin defines three different types of synchronization that must be considered in an action/object environment [Allchin83]:

1) *process synchronization:* Access to shared objects often must be synchronized in order to provide mutual exclusion. For example, solutions to the classical readers/writers problem require writers to exclude each other during the write operation. As we have seen, traditional operating system techniques for providing mutual exclusion may not be appropriate for a distributed environment.

2) *action atomicity synchronization:* Synchronizing access to objects depends not only on the current action, but also on incomplete (uncommitted) actions involving those objects. Whereas process synchronization can be viewed at the object level, atomicity synchronization requires knowledge of action dependencies.

In order to insure atomicity, either an optimistic or a pessimistic concurrency control mechanism may be used. In a pessimistic approach, concurrency control is invoked when operations are requested. Execution of the operation is delayed, if necessary, until the completion of other concurrent atomic actions. In the optimistic approach, an action is allowed to perform operations without constraint, and concurrency control is invoked at commit time. At that time, the effects of the action on its objects are evaluated. The action is allowed to complete only if doing so does not violate the objects' consistency constraints.

An optimistic approach is appropriate if concurrent atomic actions conflict with each other infrequently. Conversely, a pessimistic approach is appropriate if conflicts are more likely. The use of an optimistic scheme under such conditions would lead to an excessive number of aborts [Natarajan85].

3) *operation ordering:* In addition to the synchronization required for atomicity, it may be necessary to order the execution of object operations. For example, a queue object requires that at least one enqueue operation completes before a dequeue can be performed. This type of synchronization is necessitated by the semantics of abstract data objects. It may, like action atomicity synchronization, require knowledge of action dependencies.

Recovery is necessitated by a number of possible factors, including explicit aborts by user processes and implicit aborts caused by system failures (such as deadlock or hardware failure). As in the case of concurrency control, either an optimistic or a pessimistic mechanism may be used. In a pessimistic approach, the present state of an object may not be altered until it has been determined that recovery will not be needed. Pessimistic recovery is typically implemented using *shadowing*. An object's shadow contains the new values that are to be assigned to the object, provided the action commits. An optimistic recovery strategy permits objects to be modified, and records

sufficient information to undo the effects of an aborted action (i.e., to restore the former state of objects). In addition, the system may store information in order to redo operations. Optimistic recovery is usually implemented using *logging* The logs hold old object state information, and optionally, forward recovery information.

3.4 The Role of Processes

In the 'abstract' object model, the data structures within objects are defined as passive. 'Processes' or 'modules' are what invoke the operations of the system. The precise role of processes is not well defined in the object model itself and typically differs depending on the implementation (as will be seen in the systems discussed later in the chapter). In systems where synchronization of access to objects is provided by object managers there is a set of special processes (or modules) associated with objects that provide this function. These processes are distinct from those processes that are associated with applications. The execution of these processes can be seen as "representing" (trans)actions [LeBlanc85]. Whether application processes interact directly with objects or through some mediating processes (such as object managers) is a function of the implementation.

3.5 Active vs. Passive Objects

As previously noted, the object model defines objects as passive entities. The classification of objects in certain systems as "active" arises from one of two conditions and, like the role of processes, is implementation dependent:

(1) Objects that are automatically associated with a manager process are typically classified as active, although this represents a somewhat less clear cut case than the second condition.

(2) Processes (modules) may simply be regarded by definition as objects. This represents a departure from the original object model. It is important to note that this *does not* imply that (trans)actions are objects. Rather the processes that initiate them may be considered as such.

Having outlined the issues involved in building support systems for distributed programming, and discussed the two principle paradigms used in their construction (objects and actions), let us examine some projects currently under development.

3.6 Cronus

The Cronus Operating System, currently under development at Bolt Berenak & Newman Inc., is a prime example of the use of object-oriented techniques in building a reliable distributed operating system. Following the object model, all Cronus system activities can be expressed as operations on objects (which are organized into classes or "types"). System components such as processes, directories, and files are examples of typical Cronus objects. Each object in the system has a manager process. Because of this, and because processes themselves are defined as objects, Cronus objects are considered active. A type manager process on a Cronus host manages all objects of a given type that reside on the host. These managers, taken collectively, handle the resources represented by that type for the system as a whole. Binding of a manager to an object for a particular operation is accomplished dynamically [Schantz86].

The dynamic binding of client requests to appropriate object managers provides for efficiency and flexibility in the network context. Some objects can migrate to allow for system reconfiguration, while others are replicated to support availability. Support for location transparency permits operation invocation to be independent of the sites or the client and the object being accessed.

Much attention has been paid in Cronus to (1) providing built-in managers for standard Distributed Operating System services such as file, directory, and process managers and authentication services, and (2) creating tools for automated object manager generation. This reflects the developers' focus on the object model as the primary concern in distributed systems development:

> First, since developing application-defined objects is central to the Cronus philosophy of design, aids to the construction of new managers are a necessary tool for the programmer. Second, as the standards for new manager construction become better defined, there are opportunities for automating the generation of common parts of new managers, hence reducing the complexity and tedium of the task of coding them [Gurwitz86].

On the other hand, little has been done to develop transaction mechanisms in Cronus, beyond including transaction identifiers in messages. Transaction management and fault tolerance issues remain as areas to be addressed in future work.

It is worth noting that inter-process communication in Cronus is not limited to one particular paradigm. "While operation invocation and replies can be cast as synchronous procedure calls, they are not limited to RPC semantics. For example, Cronus IPC supports asynchronous invocations and one-to-many semantics, as well as the one-to-one semantics of RPC" [Gurwitz86].

Cronus is also independent of any specific programming language. This approach is in contrast to projects such as Argus [Liskov79] which have taken a more language-oriented approach to distributed systems development. Instead, Cronus features are accessed through subroutine calls and support is provided for multiple language interfaces. This permits use of Cronus tools in an environment and language appropriate to the application.

In the initial implementation, the kernel and support library which constitute the Cronus Operating System, run on top of the host operating system. The only requirements for integrating Cronus into the native operating system are that it support multiple processes and low level network transport facilities. As a result, Cronus is highly portable, but potentially slower than systems that are built on top of a specialized operating system kernel (see, for example, section 3.6 regarding Clouds and section 3.8 concerning TABS, which uses the ACCENT kernel).

3.7 Argus

Argus is an integrated programming language and system for the development of distributed programs currently in use at MIT. The developers of Argus have concentrated on a particular class of distributed applications. Namely, those which involve the manipulation and preservation of long-lived, on-line data. In these applications the availability of reliable distributed data is of primary importance, and real-time constraints are not severe. Examples of such applications include banking systems, airline reservation systems and distributed data base applications [Liskov85].

The Argus system provides two main mechanisms for the support of distributed computations, *guardians* and *actions*.

A guardian encapsulates and controls access to one or more resources. Guardians may be viewed as representing logical nodes of the system, since there is no direct sharing of objects between them. It is important to note that all objects within a guardian reside at the same physical node. Guardians communicate with each other as well as with user processes via *handlers* Handlers are part of the underlying communication facilities used to ensure reliable message passing. It is through handler calls that a guardian's resources are accessed. The semantics of handler calls are essentially the same as remote procedure calls, as described in chapter 2. Thus, a call message is delivered and acted upon exactly once at the called guardian and exactly one reply is returned, or the message can not be delivered and the caller is so informed.

Internally, a guardian contains data objects and processes. Processes execute handler calls and perform background tasks for the maintenance of the guardian's objects. While direct sharing of objects between guardians is not permitted, data objects within guardians may be shared by these processes.

Guardians are created dynamically and their placement determined by the programmer. In addition, Argus provides for location transparency of guardians. That is, handler calls will continue to work even if the called guardian has changed location. This allows for ease of system reconfiguration, and thus aids in recovery. Guardians themselves are recoverable entities: After a crash and subsequent recovery of the guardian's node, the support system recreates the guardian with the objects that were last written to stable storage. A recovery process is then started in

the guardian which restores volatile objects to a state consistent with that of those stable objects previously restored.

In keeping with the applications Argus is intended to support, emphasis is on maintaining consistency of data within guardians. This is the function of actions.

Actions in Argus meet all of the criteria found in database transactions, including serializability. However, these properties do not apply to all objects in the system. Objects which have the necessary synchronization and recovery properties are known as *atomic objects* Atomicity is guaranteed only when the objects shared by actions are atomic. The Argus implementation of atomic objects is based on simple read/write locking, with the usual rules: Multiple readers are allowed, but readers exclude writers and a writer excludes both readers and other writers. When a write lock is obtained a *version* of the object is created. This new version is made permanent if the action completes, and is discarded if the action fails.

In order to provide for concurrency within actions, as well as for failure containment, Argus supports nested actions (or subactions). The failure of a subaction does not force its parent action to abort. However, the commit of a subaction is dependent on the outcome of the parent. Even if a nested action commits, the failure of its parent will cause its effects to be undone.

In Argus, a parent action may not run concurrently with its children. This was done in order to simplify the locking rules [Liskov85]. The locking rules are extended to nested actions as follows: An action may obtain a read lock on an object provided that every action holding a write lock on that object is an ancestor. An action may obtain a write lock on an object provided that every action holding a read or write lock on that object is an ancestor. All locks acquired by an action are held until the completion of that action. This property of actions, together with the locking rules presented above, guarantees serializability, but accordingly limits concurrency. Given this locking scheme, it is also possible for actions to deadlock. Rather than having built-in mechanisms for preventing or detecting and resolving deadlocks, Argus relies on user processes to time out and abort actions.

Because of the emphasis on atomicity in Argus, the semantics for dealing with non-recoverable actions are awkward at best. As long as the effects of an action can be undone, the user of Argus need not write any code to compensate for the effects of aborted actions. However in cases where an action makes changes to the external environment this is not always possible. In such situations, Argus requires creation of 2 separate sequential top-level actions.

All changes to the external environment must be deferred to the second action, and are executed only if the first action commits. If the effects of an aborted action cannot be undone or if a committed top-level action has an undesired effect on its environment, actions that compensate for the problem must be defined and executed by the user.

3.8 Clouds

"The goal of the Clouds project at Georgia Tech is the implementation of a fault tolerant distributed operating system based on the notions of objects and actions, which will provide an environment for the construction of reliable applications" [LeBlanc85]. The 3 basic components of the Clouds architecture are *objects, actions,* and *processes.*

As in other object-based systems, Clouds objects represent system components, and are accessed via operation invocation. Objects are composed of four basic components: the data portion, the operation portion, the synchronization portion and the recovery portion. The Clouds object structure is pictured in figure [3.1]. Note that each object has a *volatile* and a *permanent* component. As part of the synchronization and recovery mechanisms built into the system, Clouds provides for *checkpointing* of objects; i.e. data in volatile storage is written to permanent storage. Note, too, that objects maintain per process stacks and heaps, but that there is no separate object manager process. Instead each invoking process "carries its thread of execution into the object" [Dasgupta85]. Thus, whereas objects in Cronus and Argus are active, Clouds objects are passive. It is processes that provide activity in the system. Processes can be thought of as "representing" actions [LeBlanc85].

A Clouds action is defined as a unit of work, characterized by a set of changes to objects. Actions are failure atomic; that is, an action either completes by committing or fails by aborting, and an aborted action has no effect on its environment. Part of the process of an action commit is the checkpointing of all affected objects.

> Thus the action concept successfully broadens the recovery viewpoint provided by checkpoints [of individual objects], since it encompasses *all* the changes to *any number of objects* made by an arbitrarily complex action [LeBlanc85].

Like Argus, Clouds supports nested actions for increased concurrency and failure containment. However, a number of features distinguish actions in Clouds from those in Argus: First, Clouds can support actions that involve objects on more than one machine. Thus, for example, a remote procedure call can be done without creation of a nested action. Secondly, in Clouds, a parent action may run concurrently with its children. However, an action that does so must not assume that the state of a shared object remain constant while a child is running. This was done to reduce the overhead of permitting access to shared objects only through nested actions. The following example illustrates this situation: Action A creates a file and passes it to B. A then proceeds to do other work, eventually waiting

for B to complete. This would not be allowed if B were forced to wait for A to release the lock. Instead, A would be forced to start another subaction in order to create the file [Allchin83]. Third, and most importantly, Clouds actions do not guarantee serialisability. Instead, Clouds supports a *semantics-based* synchronization scheme where the atomicity of the action is based on the semantics of the objects accessed [Allchin83][McKendry84]. The Clouds approach thus allows for breaches to serialisability when "semantically appropriate" [LeBlanc85]. In this approach, the requirements for action atomicity can be expressed in terms of each actions *view* of the object. The following examples illustrate how visibility considerations interact with synchronization of actions and object recoverability:

Suppose that a file F exists on permanent storage prior to the start of action A1. As part of A1, F is deleted, causing the storage occupied by F to be released. Although A1 has released the storage, it can not be re-used (even by A1) until after A1 has committed. Otherwise, F could not be restored in case A1 failed.

The second example concerns the traditional producer/consumer problem. In such problems, items change when they are produced or consumed. The synchronization mechanism must account for this, even though both produce and consume can be modelled as write operations. Consider actions, A1 and A2, which access a queue, Q. Assuming pessimistic synchronization, items visible to an action are those that have been added by committed actions and those entered by that action. An action can therefore remove entries that it has added, or that were part of Q's permanent state (by virtue of having been added by a completed action). Now, suppose A1 removes an item from Q. If the entry to be removed was added by the action that is removing it (i.e. A1), it will never be visible to A2. As viewed by A2 the state of Q is unaffected. If, on the other hand, the entry had been added by a committed action, A2's view of the object will change. It is therefore possible that an action will block because there are no entries in its view, even though concurrent actions are logically able to continue [McKendry84].

The notion of an action can be extended to encompass a network of related actions, or in Clouds terminology, 'work'. Clouds models work using a Petri-net notation, where transitions correspond to action executions. The state of a net is known as a *job*. The system ensures that continuity of job execution is maintained despite failures. *Job schedulers* are used to assign activities to machines in order to provide this feature.

As previously noted, Clouds differs from the other systems mentioned in implementing passive objects. This is in contrast to the use of object managers [Dasgupta85]. As a result, the role of processes differs somewhat in Clouds from that of most other systems; that is, user processes invoke object operations directly rather than through a manager process.

Another major differences between Clouds and the other systems previously discussed is the fact that the Clouds kernel was built on a bare machine and assumes no support from a conventional operating system kernel. Thus, the developers of Clouds "expect it to be more efficient and more suitable for real time applications than most other systems" [Dasgupta85].

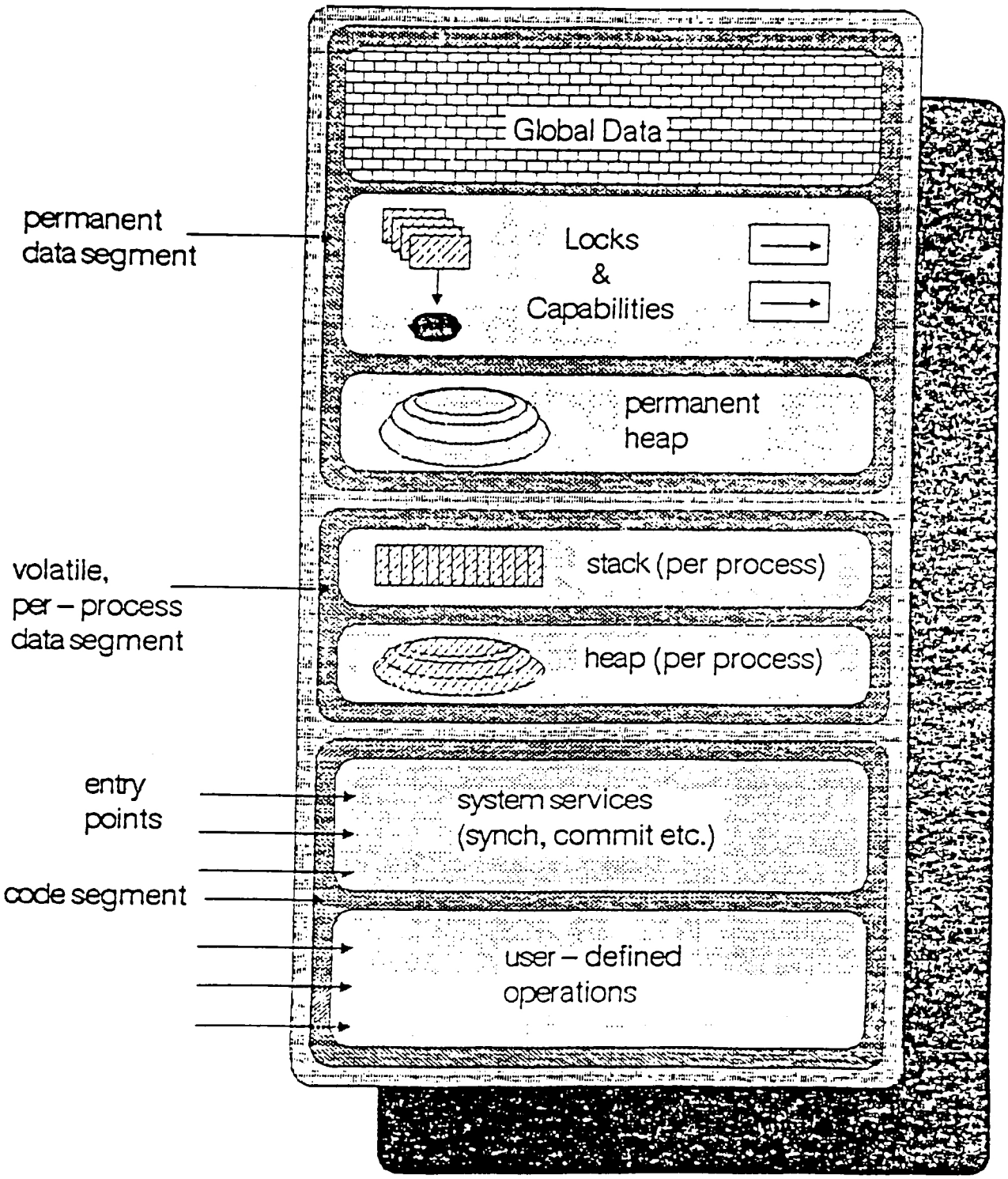


Fig. 31 Clouds Object Structure.

3.9 Archons

While Clouds uses the semantics of individual objects to increase concurrency within the action model, Archons concentrates on the relationships between objects and on the properties of transactions in which they are involved. The developers of Archons have "supplanted the serialization model with ... [a] model based on relationships among the data objects" [Sha83]. By considering the properties of transactions in an operating systems environment, they are able to achieve improved performance (through increased concurrency) without violating the consistency constraints of objects. Their *relational model of data consistency* classifies the possible relationships between data objects as follows:

Autonomous: An autonomous relationship between objects A and B implies that A can take on any value in its domain, regardless of the value of B, and vice versa. In other words, A and B can be updated separately.

Dependent: The value taken by object A is constrained by the value taken by B, and vice versa. These objects can not be updated independently.

Partially dependent: The value of object B is dependent on the value of A, only if A takes on one of a particular subset of values in its domain. "The notion of partially dependent relationships allows us to view process synchronization as the act of maintaining the data invariants [defined below] among distributed state variables" [Sha83]. For example, suppose A and B are state variables of processes P1 and P2, respectively. A partial dependency between P1 and P2 could be expressed as follows: Process P2 must enter state s1 if process P1 enters state s2, otherwise P1 and P2 may change their states independently.

In the Archons model, consistency constraints are partitioned into two parts: data invariants and action invariants. *Data invariants* are the mathematical representation of the dependency relationships among data objects. Data invariants must be preserved by all processes or transactions. *Action invariants* represent consistency constraints enforced by individual transactions in addition to data invariants. Action invariants must not violate data invariants. For example, let A and B be data objects with data invariant A >= B. Transactions T1 and T2 may each have their own action invariants: e.g. "set A equal to the current value of B" and "set A = (B+10)". These invariants must hold at the end of the transactions, but may not necessarily hold at other times.

The following definitions are also key to understanding the Archons model:

Atomic data sets are user-defined disjoint sets of data objects, with their corresponding set of data invariants. For example one data set may have data objects A and B with invariant "A=B" and other objects C and D with invariant "C > D".

Conformity is defined as a concurrent access to shared data objects that preserves all data and action invariants.

In order to achieve increased concurrency without violating data invariants, Archons replaces the global serialization required by the database model with setwise serialization enforced only when it is necessary. Thus, the relative ordering requirement of serializability is not enforced when the relation among data objects is known to be autonomous. For example,

Let A,B, and C represent the number of jobs on three machines. Let transaction T(i,j) represent the transfer of a job from machine i to machine j with the corresponding changes to the job counts on each machine. Consider the following sequence of transactions T(A,B), T(B,C), and T(C,A) that update A,B, and C: Transaction T(A,B) precedes T(C,A) on A, T(B,C) precedes T(A,B) on B, and T(C,A) precedes T(B,C) on C. The concurrent execution of these transactions would violate the relative ordering requirement of the serialization model. However, the Archons model allows this sequence of transactions, because the number of jobs on each machine is independent of the number of jobs on the others (i.e. A,B,C are autonomous) [Sha83].

The relational data model provides the foundation for *co-operating transactions* Co-operating transactions are transactions that communicate with each other and satisfy the conformity condition; i.e. the execution of concurrent transactions is defined to be correct if it satisfies both the data and action invariants, independent of whether the transactions are serializable. Thus, the relational model of data consistency, coupled with co-operating transactions, permits increased concurrency without violating invariant conditions.

The main weakness of the Archons model is that interactions between data objects must be known a priori. While this may be a valid assumption for certain operating system objects, such as job queues, it is not acceptable for general purpose distributed applications.

3.10 TABS

TABS is an experimental system, developed at Carnegie-Mellon University, that provides operating-system level support for distributed transactions [Spector85].

The two main components of the TABS *model* are transactions and objects. Transactions are initiated by processes and invoke operations on objects. In order to facilitate parallel execution of transactions on multiple objects, and to permit portions of a transaction to abort independently, TABS supports a subtransaction facility. This facility can be characterized by its synchronization and commit policies:

- A subtransaction behaves as a completely separate transaction with respect to synchronization.

- Subtransactions may not commit until the parent transaction commits, but they can be aborted without causing the parent action to abort. This is useful if a transaction can tolerate the failure of some of its operations. That is, such operations can be executed as subactions.

Objects in TABS are instances of abstract data types and are encapsulated in processes called *data servers*. Data servers execute operations on behalf of transactions. In addition to executing operations, data servers also contribute to the synchronization and recovery of transactions. Data servers support process synchronization using locking. Type-specific locking, based on operation semantics, is used to achieve increased concurrency. Each operation on an object requires that a transaction obtain the appropriate lock. A lock compatibility relation is used to determine whether the lock may be acquired.

The TABS *implementation* consists of seven basic components, as shown in figure [3.3]:

The *Accent Kernel* supports processes with private virtual address spaces. These processes communicate via message passing using a port-based mechanism (as described in Chapter 2). The Accent kernel guarantees that messages are delivered in order and at most once.

Application processes initiate transactions and invoke operations on data servers.

Data Servers encapsulate two kinds of data: objects and synchronization information, and three kinds of code: code to implement operations, code to implement recovery of objects, and code for type-specific synchroniza-

tion. A data server may encapsulate one or more objects, which in turn may be of more than one type. This allows for greater parallelism within the servers. In many ways, data servers correspond to guardians in Argus and like guardians are recoverable entities [Spector85].

Name Server processes map names used by application processes to particular objects within data servers.

The *Recovery Manager* maintains the logs and presides over transaction abortion and recovery. During normal operation the recovery manager receives records from data servers, the Transaction Manager, and the Accent kernel and writes them to the log. Log records are written in the order that the associated messages are received, thus providing for serializability of log updates. During transaction abort and recovery, the Recovery Manager reads the log and sends log data to the appropriate system components. Components may be required to redo or undo their operations in response to these messages.

Each node in the system has a single *Transaction Manager* process which coordinates the initiation and termination (commit or abort) of transactions. Accordingly, each Transaction Manager keeps status information for every transaction active at its node. This information is provided in the form of messages from processes, data servers and the Communication Manager. The Communication Manager sends a message to the Transaction Manager the first time a local process sends a message to a remote server. At this point the transaction manager becomes aware that remote sites are involved in the transaction, but it does not know which ones. Only during commit processing does the Transaction Manager acquire the complete information (spanning tree) for the transaction.

The *Communication Manager* works via the ACCENT kernel to provide transparent node-to-node communication. The communication manager also plays an important role in coordinating transaction commit and abort. It makes use of the transaction identifiers that are included in messages in order to construct the local portion of the spanning tree used by the Transaction Manager during two-phase commit. In this respect, TABS is clearly unique: That is, it uses the communication facilities as a means for "registering" remote sites in a transaction. In addition, the communication manager provides network monitoring. In conjunction with data-servers and application processes, it detects communication failures and helps in the detection of node crashes. There are three ways in which this is done in TABS:

(1) Loss of Connection: This indicates the crash of a node with which the node is communicating. It is detected by the communication manager either through a negative acknowledgement indicating that the remote site no longer knows of the connection, or through the lack of any acknowledgement.

(2) Time-outs: Time-out while waiting for a reply message indicates a problem with a remote data server or application process.

(3) Relationship Mismatch: This mechanism is another way of detecting node crashes. In it, a per-transaction information is used to check for inconsistencies between local and remote nodes. Each node maintains a per-transaction table that records what relationship the remote nodes have to the local one. When a node crashes, its tables are lost. Subsequent messages from transactions that started prior to the crash will encounter a relationship mismatch.

One of the strengths of the TABS implementation is the modularity (and thus flexibility) designed into the system. At the same time, the system pays the price in terms of cost of communication among components. "A future reorganization of the system could remedy this shortcoming by including more of the functions of the Recovery, Transaction and Communication Managers in the Accent Kernel" [Spector85].

Extensions planned for TABS include support for detection and resolution of deadlocks. According to its developers, "deadlock detection will be difficult in TABS because data servers maintain their own lock information" [Spector85]. Plans call for implementation of a deadlock detector which would interface with data servers to obtain lock information. Some deadlock information could also be gained from knowledge of which processes are awaiting messages on which ports. (Yet another job for the communication manager).

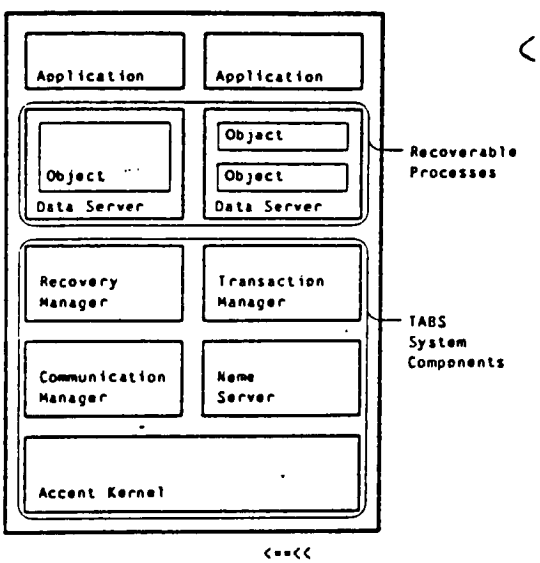


Fig. 3.2 The basic components of a TABS node.

3.11 Real-time Process Control

The systems described thus far have not been concerned specifically with real-time applications. Although Cronus, Clouds, and to some extent, Archons and TABS may be applicable to process control applications, they have concerned themselves primarily with objects commonly found in operating systems; i.e. abstract data items such as directories and queues, and to a lesser degree, physical devices such as printers and disk drives. Following a brief description of the basic requirements for real-time control systems, I will discuss a system built expressly for distributed process control.

The requirements for distributed process control systems differ in a number of ways from those of database and operating systems previously discussed, though the issues of reliability, concurrency control and synchronization, and structuring of communications remain. These differences manifest themselves primarily in the following areas:

- Concurrency: In real-time systems, many sensors and actuators must be controlled in parallel and their actions properly coordinated. There must be real-time synchronization of the participants in transactions (as opposed to event ordering, as required by databases and operating systems).

- Communication: Communications concerned with the direct physical operation of the process are highly time sensitive. Communication delays must be kept small as compared with typical database and operating systems applications, with a predictable maximum delay. On the other hand, logging messages and statistical data gathering are much less time critical.

- Reliability: The reliability of direct physical operations is also critical, since an error in a process control environment clearly presents a much greater safety hazard than in databases or operating systems. A greater degree of redundancy is therefore likely to be needed in real-time systems.

- Recovery: In databases and operating systems recovery is linked to atomicity. In real-time process control, atomic actions are frequently not applicable. Actions that concern the direct physical operation of external devices are often "irrevocable" in the sense that undoing them does not restore the previous state of the system. Similarly, once such an action occurs the view of other system components is immediately affected. These actions are therefore neither failure nor view atomic. In the event of erroneous actions (essentially the

logical equivalent of an aborted transaction), special actions unique to the application must be taken to compensate for their effects. As a perhaps somewhat extreme example, consider actuating a controller that launches a missile, and suppose further that this action is an undesired event. This action may (hopefully) be compensated for by some special handler module (that, for example, changes its trajectory), but it is clearly not atomic.

- System Configuration: Control systems are relatively stable, especially in comparison to operating systems, in the sense that there is less need to create and destroy tasks dynamically. However, reconfiguration of process elements is an important requirement. Reconfiguration is necessitated by a number of factors involving both hardware and software, including component failures, technological improvements in components, better automation, or restructuring of the process itself based on a different analysis of operations [Hommel85]. As changes usually affect only part of a distributed system, reconfiguration should not disrupt the whole system control process.

3.12 CONIC

The CONIC system is currently being used by the Mining Research and Development Establishment in Britain to produce software for underground monitoring and control systems. Its use in other distributed control systems is under investigation [Kramer85].

Following Kramer, we will define a Distributed Computer Control System (DCCS) as one where "the control function has been partitioned into subfunctions which can be implemented by a set of physically distributed computer stations. Typically a subfunction will be concerned with the local control of an item of a plant or a machine" [Kramer82].

A major objective of the CONIC architecture is to "separate the concerns of writing individual software components (programming-in-the-small) from those of constructing or configuring a system...(programming-in-the-large)" [Kramer82]. These notions correspond in the object model to writing modules to implement objects and defining the relationships between objects in a system.

In CONIC, a *module* is "the software abstraction of a station" [Kramer83]. Modules perform local monitoring and control functions, and are usually associated with a single device. Module instances form the basic building blocks from which the system is configured. As such, a module is the smallest software component that may be distributed or replaced. In order to synchronize their actions, modules communicate by message passing. The interface to a module consists of typed *entry* and *exit ports*. Entry ports are used for receiving messages and exit ports for sending. The only exception is in the case of a "request-reply" transaction in which the reply is received on the same exit port that the request was made, and the reply is sent on the same entry port on which the request was received. An outgoing message is directed, not to the entry port of a receiving module but to an exit port of the sending module. The recipient(s) of the message is (are) therefore determined by the connection of ports. Binding of a message to its destination occurs during the system configuration stage.

As part of the configuration of an application system, modules are connected by *linking* of entry and exit ports. The linking of modules defines the relationships between them within the system. CONIC provides a Configuration Language for specifying connections and thus for building systems. The following interconnection structures are

supported:

(1) One-to-one: Bi-directional communication between two specific components occurs with the linking of one exit port to one entry port.

(2) One-to-many or multicast is achieved by linking one exit port to multiple entry ports.

(3) Many-to-one: One at a time access by many modules to a server module is provided by linking many exit ports to one entry port

Although the configuration description may be written at any time, module linking is the last phase prior to actual startup in the initialization of the system. The complete steps for the process are shown in figure 3.3 . One of the products of the system configuration phase is a configuration description file that can be used view and/or modify the interconnections between system objects.

The CONIC Operating System supports and manages the execution of the DCCS through the following functions:

(1) Module Management -

(a) Downloading of module code into station storage

(b) Creation/deletion of module instances

(c) Start/Stop of module execution

(2) Fault Management -

(a) Detection and reporting of module program errors and station hardware failures

(3) Connection (Communication) Management -

(a) Linking/unlinking of ports

The operating system is constructed using the same module structure as the application system. CONIC services are made available through entry ports. Services may be invoked locally or remotely through message passing.

As we have already noted, there is a need for dynamic reconfiguration in DCCS in order to extend or modify the system or to handle failures. In view of these requirements, configuration description and reconfiguration

management play a central role in the CONIC system. The *Configuration Manager*, in conjunction with the CONIC Operating System, provides for on-line system modification. The Configuration Manager translates requests to change the system, expressed in the CONIC Configuration Language, into commands to the operating system to execute reconfiguration operations. It also validates the change specifications against the current state of the system. A configuration may be modified using the commands start/stop, link/ unlink and create/delete. These may be executed while the rest of the system is operational.

There are certain limitations to the CONIC system which, while they may be appropriate for process control applications are less than desirable for general purpose distributed systems. First, the CONIC system does not allow for dynamic migration of operational components between stations. The motivation for not providing this facility stems from the nature of DCCS applications: "Stations are typically located with the sensors and actuators they serve. Migration of software function makes little sense when the hardware function cannot be moved" [Kramer82]. As a result, fault-tolerance can only be achieved through replication of components or by reconfiguration after failures. Secondly, CONIC does not automatically save the state of system components. This would have to be provided for by explicit checkpoints programmed into the application software.

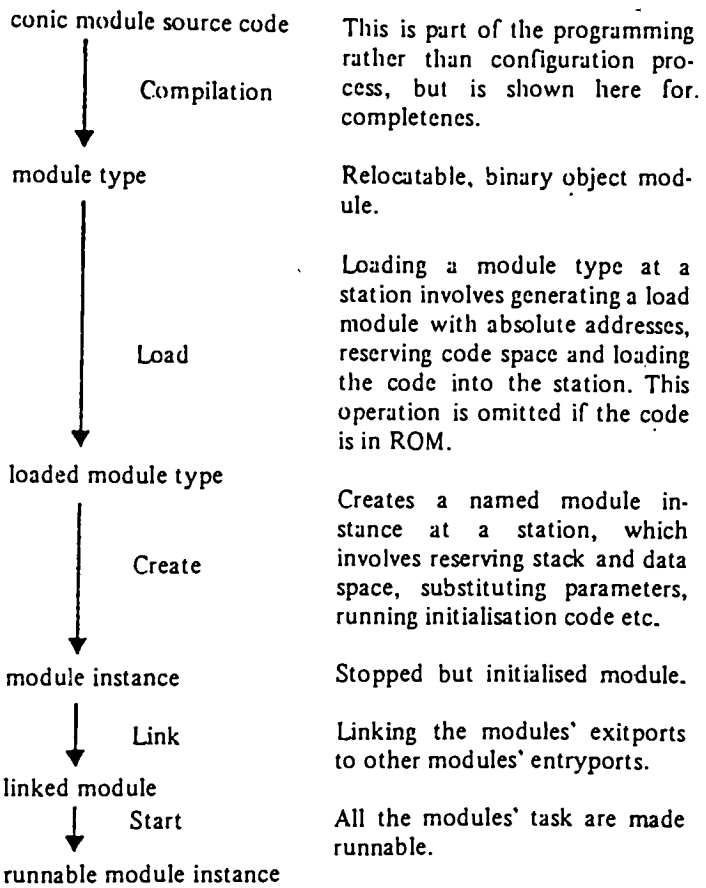


Fig. 3.3 *Stages in installing a module at a station* — CONIC

Chapter Four

Activity System

The Activity model was originally developed by Carla Ellis, Jerome Feldman, and James Heliotis at the University of Rochester and detailed in a thesis by Dr. Heliotis in [Heliotis84], but as yet no implementation of an activity system exists. This chapter describes the basic components of an activity system, and presents an example to illustrate how activities might be used. The chapter concludes by contrasting activities with systems previously discussed.

As in other object based systems, an activity system *object* is a system entity that maintains an internal state and an abstraction of its function that is made available to the outside world via a communication interface. In an activity system, objects communicate using message passing. The system assumes port-based communication (see Chapter 2) [Heliotis84]. Objects may be active (e.g. process modules) or passive (e.g. files). This is in contrast to other systems previously discussed in which objects belong exclusively to one of the two categories.

Each object is identified by a unique *object id* Although it was not proposed as part of the original activity system design, a name server could be used to map user supplied names to object id's.

Also associated with each object is a *role* , a tag used to classify objects according to their purpose in a specific application. The role provides additional semantic information about the object (beyond its id and possibly its name).

Object implementors are responsible for the creation and destruction of objects of a given type and as such are similar to Cronus type managers. For passive objects the implementor also performs the actual operations of the object.

Each object also has associated with it an *object manager* that provides interpretation of activity commands for the object, as opposed to application related commands used to invoke the object's operations. In this sense, the object manager is akin to the synchronization and recovery portion of Clouds objects, or TABS data servers.

Modules are the basic built-in object type provided by the system. All other types are user-defined and are implemented by modules that act as the object implementor for that type. A module consists of data and code for manipulating the data, together with a message interface for communicating with the 'outside world'. Modules constitute the active objects in a computation, and are thus similar to Clouds or TABS processes.

Module implementors are responsible for the creation and destruction of modules. As the message interface is part of the definition of a module, the implementor also handles initiation of module communication. In the event of the death of a module, it is the implementor that notifies other interested parties via the activity coordinator (described below).

The formal definition of an *activity* is "a dynamic identifiable collection of state information that is spread among a dynamically changing set of objects in a computational network" [Heliotis84]. It is important to note that an activity is not a new type of object. Rather, an activity represents a relationship between objects. For objects to participate in the same activity implies that there is some logical connection between them; that is, they share a common purpose within a computation. Thus, an activity consists of "a dynamically changing set of objects that cooperate to achieve a single goal" [Heliotis84].

In order to provide a "handle" on activities in the system, each is identified by a unique *activity tag* . Activity tags are useful in a number of areas:

Communication - One of the main uses of activities is that one may issue a command to all objects performing a specific task, *without* having to to first obtain a list of those objects and establish communication with each of the corresponding managers individually. In order to do so, however, objects must be registered in the same activity. Typically, if two objects communicate then they are logically connected in some way and ought therefore to belong to the same activity. Whereas in TABS such registration is automatic, the activity system does not mandate tagging of communications. Thus, short-lived message exchanges do not require the additional overhead of establishing an activity context.

Code - Activity tags may be used to establish an activity context for a block of code. With it, code may identify on whose behalf it is currently working. If work on a particular activity is suspended or terminated, the object may continue with others.

Data - The sharing of objects requires that data structures be "indexed" by activity. This permits updating of shared objects on a per-activity basis. Furthermore, activity tagging of data could be used in conjunction with knowledge of activity structure to determine relations between objects as in the Archons "relational data model".

Sub-activities represent smaller logical sub-goals within an activity. By partitioning sub-activities such that they access disjoint sets of objects, one may be able to identify good candidates for concurrent processing. This is essentially the approach used in Archons' "cooperating transactions". Sub-activities also provide for failure containment, in that objects participating in one sub-activity are isolated from problems occurring in others. The need for an atomic step within an activity is another reason for creating a sub-activity. In this case, sub-activities resemble nested atomic actions. However, it is important to note that there is no atomicity requirement for (sub-)activities. In fact, this is the main difference between activities and most of the other systems we have discussed. The resulting structure of an activity can be viewed as a tree, where internal nodes represent sub-activities and the leaves represent objects registered in their parent activities.

Each activity in the system can be represented by an *Activity Control Module* (ACM). Just as an object manager responds to activity commands for its object, an ACM handles activity commands for each of the activity's sub-activities and objects. The difference between the two types of managers with respect to objects is as follows: The object manager understands the implementation of the object but has no knowledge of the purpose of the object within the context of the activity. Conversely, the ACM knows the purpose of the object with respect to the activity, but knows nothing of the object's implementation beyond its interface description.

Although an object may participate in several activities, each has exactly one *owning activity* The object is created within it, and does not outlive it. The owning activity plays the primary role in controlling the object: It is possible, for instance, to design the support system so that certain operations on an object are only allowed if done in the context of the owning activity [Heliotis84]. For example, killed by whom. It may well be that this right should be restricted to the owning activity.

The *Activity Coordinator* (AC) acts as a kind of "clearinghouse" for activity-related information and communication. Its functions are somewhat akin to those provided by the Transaction and Communication Managers in

TABS. In addition to generating activity tags and object id's, the AC is responsible for the following:

(1) Activity Database Management

(2) Activity-Related Communication

Activity Database Management

The coordinator maintains a database of information pertaining to objects and activities. At present, this information consists of the following:

For objects -

(1) The owning activity,

(2) a list of activities in which the object is registered,

(3) its role

(4) the ports of the object manager and implementor,

(5) various flags used by the AC for routing communications.

For activities -

(1) The ACM port,

(2) the activity tree structure (i.e. parent activity and sub-activities),

(3) a list of objects registered in the activity,

(4) the status of the activity (e.g. Suspended, Committed, Aborted),

(5) various flags used by the AC for routing communications.

Note that as further uses of activities become apparent, additional information may be required as part of the database. The current database therefore represents somewhat of a "bare-bones" implementation. For example, it is possible that activities could be used, as in TABS, to provide communication monitoring. In this case, the AC would maintain a "connection status" for each object and ACM in an activity. Each activity that involved remote sites could then be registered in a "communication monitoring" activity, with communication failures being reported to an appropriate handler module.

Activity-Related Communication

The coordinator handles the routing of activity related messages. There are 4 types of commands handled by the coordinator, each with different protocols for propagating the corresponding messages:

(1) Activity Commands - are sent to each of the sub-activities and objects in the activity. The command is executed recursively for each of the sub-activities, (thus effecting a bottom up order). Next, the appropriate object managers receive the command. Each sub-activity and object manager/implementor must reply within a timeout period specified in the command. In the original thesis on activities, failure is assumed "if no reply comes within a reasonable amount of time" [Heliotis84]. As 'reasonable' may well be quite different for a database application and a real-time application, I believe that the timeout requirement should be included as a command argument. Activity commands include creating and ending activities, and removing objects (either by "peaceful" or "violent" destruction [Heliotis84]).

(2) Object Commands - are sent directly to object implementors/managers, and have to do with the creation, destruction, registering and de-registering of objects in activities. For these commands, no activity tree search is needed.

(3) User Commands - may be sent via the activity coordinator to any ACM, object implementor or object manager. These commands are likely to be application specific, and thus the coordinator is not expected to have any semantic understanding of them.

(4) Emergency Notices - are sent whenever an object fails irrecoverably. In this case, the implementor will first destroy the object, and then notify the coordinator. The message will then be sent to all ACMs for activities in which the object was registered. Emergency notices are also generated in the event an ACM fails. Since there is presently no mechanism for recovering ACMs, this results in a failure of the entire activity. Notice of ACM failure is sent via the coordinator to the parents of the ACM.

As with the information stored in the activity database, this set of messages represents something of a minimal working set that can be augmented as new uses of activities arise. For example, it may well be that certain commands frequently defined by users may be incorporated into the "standard" set of activity and object commands.

4.2 A Sample Activity

The following example illustrates a potential use of activities: Consider formatting and printing several small text files that make up a larger report as, for example, chapters in a thesis. Suppose further that this activity is to take place in a network, where the text files exist on a personal workstation, a grammar analyzer and text formatter reside on a shared computer, and there is a remote printer that is in turn shared by several machines. As part of the processing, the text files are copied to the shared machine where they processed first by the grammar analyzer and then by the text formatter. Note that several chapters may be processed in parallel, and that they may be in different stages of processing. Moreover, with respect to each chapter, grammar analysis and text formatting may take place concurrently. After each chapter has been formatted, it is printed on the remote printer. In addition, suppose that the user who initiates this procedure wishes to be notified of certain events, such as the success or failure of a file transfer or the occurrence of a certain number of repeated words in a file. This activity involves several shared server modules (file transfer, grammar analyzer, formatter, print server) and passive objects (text files and printer) on a number of machines, as well as the user (who could be considered as an active system object). The overall activity (creating a report) is composed of a number of sub-activities defined by the processing of each text file. The processing of each chapter represents a sub-activity that is further composed of sub-activities, both atomic and non-atomic. The sub-activity "transferring a text file" would most likely be defined as atomic, since we would probably not want to process partial chapters. On the other hand, printing of the files would clearly *not* be atomic, since it can not be undone and re-done nor can its effects be "hidden" from other activities. In this example, the "success" or "failure" of the grammar analysis activity could be determined by the user for each file and the formatting activity continued, suspended or aborted accordingly.

4.3 Activities vs. other recent systems

In distributed system management "there is a distinction between what can be done automatically, independent of the particular problem being solved, and what requires application-specific policies" [Ellis85]. Recent related work has focused on that which is automatic (through use of atomic actions), without providing a means for specifying user-defined synchronization and recovery mechanisms. The activity system approach "differs from similar research efforts by providing additional flexibility for application specific ... control features" [Ellis85].

As we have seen, most of the recent projects have taken a similar approach to the development of action-based systems; that is, they have attempted to make transactions more general-purpose by modifying the existing model to eliminate unnecessary serializability constraints. These systems have focused on providing atomicity as a primary goal. Activities, on the other hand, provide a more generalized approach to structuring of distributed computations. The primary goal of activities is to be able to establish logical connections (relationships) between objects that are appropriate for a given application. If atomicity is the principal requirement of an application, then activities can be used to achieve this. However, and this is especially true for real-time systems, not all object interactions fit this paradigm. As we have seen, actions involving external physical devices are often "irrevocable" and require special actions to undo their effects when necessary. As a result, such actions can not be atomic. An activity-based system is well suited to actions involving external devices because (1) it does not assume atomicity and (2) it allows users to define appropriate responses to failures or erroneous actions.

For error recovery most of the systems previously discussed use some variation of (nested) atomic actions with a two-phase commit protocol. With an activity system, no particular recovery algorithm is mandated. Instead the programmer is provided with tools for specifying recovery actions appropriate to the application. Thus, for example, nested atomic actions could be implemented *within* the framework of activities. By the same token, different mechanisms are possible. In each of the systems that support nested atomic actions, the actual commit of a subaction is contingent upon the commit of the top-level action. In an activity system, the user is able to supply alternative actions that cause the top-level activity to "prematurely" commit some or all of its sub-activities. As a result, the changes made by those subactivities become permanent regardless of the subsequent abort of their ancestor

[Ellis82].

Another exception to the atomic action approach can be seen in the CONIC system. In this system, connection of modules forms the basis of inter-process communication. It can also be seen as defining logical relationships between objects. In the former sense, CONIC is closer to the communication-oriented systems described in chapter 2 and shares the same weaknesses. In the latter sense, the configuration descriptor file created for linking modules represents a kind of "road map" to the logical relations between system components. However, in contrast to an activity system, this provides no means for direct manipulation of a group of related objects. Synchronization and recovery mechanisms must still address each module separately. In activities, a more cohesive plan of action may be effected, because activities address all related objects. Unlike an activity, the CONIC configuration description does not truly provide a context for handling of failures or process synchronization.

In addition to providing alternative to atomic actions for recovery and synchronization, activities can be used for other types of distributed computation management.

> One major advantage of the activity concept is that it provides a *contezt* for the handling of faults that may arise in a distributed computation and *for other kinds of dynamic control.* [italics mine] [Activities]...can be used to obtain atomicity through a commit protocol involving the objects in the context of that activity. However, atomicity is not a central focus of the model [Ellis85].

For instance, activities can be used to monitor the current status and progress of a distributed program. This can be especially useful in detecting or preventing deadlock. The context information provided by activity tags also allows for selective tracing and debugging of related modules.

Chapter Five

Implementation

For this thesis, I have implemented a portion of an activity system, namely an Activity Coordinator (AC), together with other "servers" and object modules required to produce a demonstrable Activity System. This system provides an environment in which to test the AC, as well as to gain some insight into activity-based programming. In this chapter I will describe the functional capabilities of the Coordinator, along with the design for each of the 3 implementation "phases". The simulation and test system will be described in Chapter 6.

5.1 Activity Coordinator Functions

The coordinator is able to process the following commands. Those marked with + are additions I have made; all others are from the original Activity Model thesis [Heliotis84]. A "manual" entry for each of these functions is found in Appendix A.

(1) Create an object - The "user" requests creation of an object via the Object Implementor (not to be confused with the implementor of a particular object – see Chapter 6). The Object Implementor in turn supplies the AC with manager and implementor ports for the object, a role, and a tag for the owning activity. The AC creates a new id for the object and registers it in the owning activity. The AC then sends a "Register Object" command together with the new id to the implementor and manager ports supplied by the user. Note that under this arrangement it is the responsibility of the user to arrange for the creation of objects via the Object Implementor. The alternative would be for the user to request creation of the object through the coordinator without providing the ports. The coordinator would then forward the request to the Object Implementor for that object type and return the ports the user.

(2) Register an object - This command is used to register an existing object in an activity other than its owning activity. The request contains the id of the object and the tag of the activity in which it is to be registered. The AC adds the object to the activity's list of participating objects after passing the "Register Object" mes-

sage to the object's manager and implementor and receiving a positive reply from each of them.

(3) Deregister an object - like "Register" except that the id is removed from the list of objects participating in the indicated activity.

(4) Remove an object - results in the "peaceful" destruction of the object. The AC sends a "Deregister" command to the object manager and implementor for all activities in which the object was registered. If a positive reply is received, the object id is invalidated.

(5) Destroy an object - similar to Removing an object except that the AC sends "Object Died" notices to the ACM's, in addition to the Deregister command sent to the object manager and implementor. Also, the id is invalidated regardless of the replies received. Typically, the sending of the deregister command will fail, because the object in question has already been destroyed. The reasons for this are based on the implementation of objects (see section 6.5.5).

(6) Object Died - This message is sent by the AC to the ACM of the owning activity as the result of the "violent" destruction of an object.

Each of the following "get" functions (9-13) is simply a lookup operation in which the AC returns the requested information to the user. It is assumed that the user is entitled to this information by virtue of being able to refer to the object/activity by its id/tag. For future AC implementations "security checks" could be performed on these requests, based on capabilities lists associated with each object/ activity.

(7) Get an Object's Implementor Port

(8) Get an Object's Manager Port

(9) Get an Activity's ACM Port

(10) Get an Object's Owning Activity Tag

(11) Get an Object's Role

(12)+ Get an Activity's Status

(13)+ Get an Activity's Parent Tag

(14) Create a New Activity - The user supplies either the tag of the activity of which the new activity is to be a child or 0 for creation of a new root activity, and an ACM port. The coordinator creates a tag for the activity and returns the status to the user.

(15) ACM Failed - When an ACM fails, the AC sends ACM Failed notices to the parent activity and a terminate message (see below) to the sub-activities.

(16) Activity Terminated - This message is sent to the ACM of the parent activity as the result of the normal termination of an activity.

(17) Send Activity Command - causes a command to be propagated to all sub-activities and objects in the tree rooted at a specified activity, as well as to the activity itself. The message is sent to each of the sub-activities recursively, effecting a bottom-up order. The message is then sent to the implementor and manager of all objects registered in the activity. Objects and/or activities that are flagged as being "handled" (see command #18) by a parent activity do not receive the message. System defined activity commands include "Terminate", "Suspend", and "Free". In addition to routing of messages, activity commands require the coordinator to update the status variable associated with activities. The "status" of an activity is simply an integer value associated with the activity's database entry that reflects the last activity command received by its ACM. It is presently of no use to the AC itself, but rather is intended to provide additional information to the user (see below, re: atomic actions).

The following activity commands are recognized by the system:

Terminate - Upon receiving notice that the ACM has processed the command, the coordinator invalidates the activity tag and sends an "Activity Terminated" notice to the parent activity.

Suspend - Work on behalf of the activity is halted but not aborted. The AC updates the status variable of the activity.

Free - This command is the opposite of Suspend: Work on behalf of the activity is resumed. Again, the AC updates the status variable of the activity.

If atomic actions were to be incorporated into the system, the built-in activity commands would be amended to lude "Sync", "Commit", and "Abort". "Sync" would be used to pre-commit sub-activities, where success of the ımand indicated that sub-activities were prepared (ready to commit). "Commit" and "Abort" could be used as programmer saw fit to implement atomic transactions; i.e. there would not be any imposed semantics. The AC uld simply be required to update the status of activities based on the new options. It is here that I believe the ivity status variable might well be of use to the programmer. User defined activity commands may be sent to)activities and objects in a similar manner. Under the current implementation, the user assigns an integer value ater than 4096 to associate with the command. This message number together with any additional data (up to 248 :es) constitute the activity command itself. The message sent to the AC by the user is as follows: :ND_ACT_COMMAND message_number additional_data destination_tag] The activity command (i.e., message mber + additional data) is sent to the indicated activity's subactivities and objects as described above. The only :eption is that, within the present implementation, these messages can not be marked as being handled.

(18) ACM "Handles" Messages - blocks one or more participants in an activity from receiving activity commands. This command may be applied to a sub-activity, and/or object. The coordinator marks the corresponding sub-activity/objects as not receiving a particular command. This is done using a bit mask (associated with each sub-activity/object) to select which commands are to be blocked.

5.2 Activity Coordinator Design

Figure 5.1 shows the functional organization of the Activity Coordinator (together with the file names in which : corresponding code is found). The Activity Coordinator is organized as follows: There is a process, known as the vitchboard", that listens at a well known address. The "switchboard" accepts connections from "users" (modules |uesting activity-related services) and spawns a message handling process (the "msghandler") to service the |uest. The "switchboard" passes the connection to the message handler, and is then free to process any other |uests. The message handler in turn mediates further communication between the user and the AC. The message ıdler parses the request and, depending on the operation required, invokes the appropriate operation handler

("ophandler") routine. It is this "ophandler" routine that performs the bulk of the actual processing. The handler performs the following functions:

- it controls communication between the user and the managers/implementors: After checking the validity of the requested operation, the request is forwarded to the parties involved in the activity. (At this point, the only "validity" check is that the object/activity referenced exists). The handler then waits to gather replies from each destination manager/implementor. Based upon these replies, it (1) notifies other "interested parties" of the results and/or (2) returns the appropriate result to the user.

- it updates the activity database: the request may involve the creation or destruction of objects/activities and the corresponding modifications must be made. Note that these updates should take the form of atomic actions. In addition to returning a status value (0 for success, -1 for failure), the "ophandler" routine also supplies the "msghandler" with the reply message that is returned to the user.

Figure 5.2 illustrates the event sequence for an activity system operation.

Implementation of the Activity Coordinator was originally intended to be done on SUN workstations running 4.2BSD UNIX(tm), however the final implementation took place on AT&T UNIX(tm) PC's running System V Release 2. As part of the design, I attempted to make the system compatible across both versions of UNIX(tm). This goal dictated certain design and implementation decisions and accounts, in part, for some of the inefficiencies of the current system. Dependencies on a particular version of UNIX(tm) are noted where applicable. For inter-process communication the WIN/3B Socket Compatability Library (Wollongong Group) [WIN85] was used.

Implementation was done in 3 phases: The first version consisted of a single centralized AC with one "switchboard" process and one "msghandler" process maintaining a single copy of the database. This simple prototype AC provided ease of implementation and allowed for debugging and refinement of communications with other activity system elements. The weaknesses of this initial AC are rather obvious and its lifespan was kept mercifully short. If the machine the AC is running on goes down, or becomes isolated by failure in the network, the entire activity system ceases to function. Even presuming that this does not happen, the coordinator represents a major communications bottleneck, since all activity messages must pass through one machine. The bottleneck problem is compounded by the fact that the AC runs only one message-server process and maintains only one copy of the activity database,

thus failing to take advantage of possible concurrency.

The second version of the AC provided a greater degree of parallelism within the Activity Coordinator's database, while remaining as a single centralised server. In this version the AC processor spawns multiple handler processes that access a shared database. Since these processes are located on the same machine, it would have been advantageous from the standpoint of response time to implement the database using shared memory. Such facilities are available in System V.2 UNIX (tm) [ATT85], and SunOS V3.2 and higher [McManis87]. Both provide shared memory segment and semaphore operations that could be used to create and manipulate common "C" data structures and thus provide a shared memory version of the database. However, as 4.2 BSD UNIX (tm) provides no such mechanism for memory sharing between processes, the database was implemented using shared files. While this implementation is slower than one using shared data segments, it is more fault-tolerant, since the database is in "permanent" storage. The UNIX(tm) file system provides facilities that allow processes to synchronize access to shared files. Under 4.2 BSD UNIX(tm), locking is provided at the file level using the "flock" system call. Two types of locks are supported: shared (for readers) and exclusive (for writers). Locks may either blocking or non-blocking with the former as the default. If record level locking is required under 4.2 BSD a separate lock management server must be implemented. Under System V UNIX(tm) record level locking is supported, using the "lockf" system call. Only exclusive locks are available under System V. Locking calls from processes that attempt to lock a previously locked section of a file will either return an error value (non-blocking) or be put to sleep until the resource is available (blocking). Blocking calls to "lockf" are scanned for deadlock by the operating system before the process is set to "sleep" on the locked object, and thus the use of "lockf" is guaranteed to be deadlock free. These facilities are used by the C-Tree file management package to maintain B-tree indexed shared database files. This is described in more detail in section 7.1.3.

In the final version of the AC I sought to alleviate the communications bottleneck associated with a single centralized server, by distributing the AC over all machines in the system. This version employs a partitioned database, where activity/object records are distributed as follows: For non-root activities, the host machine of the parent activity becomes the repository for the activity's database entry. Otherwise, the machine on which the activity is created holds the database record. This node is then considered the "home" machine for the activity, and the tag of

the activity will so indicate. Note that this notion of "home" machine is independent of the machine on which the ACM resides. On the other hand, object database records are stored on the host on which the object is created, regardless of the home of the owning activity. As in the case of activity tags, the id will indicate the home machine of the object. Under the current Object Implementor, the implementor and manager ports for the object will reside on the same machine as the object (see section 6.3.3 re: Object Creation).

Under this arrangement, the AC must provide for message forwarding when the request pertains to a non-local activity or object. This implies that the switchboard must have some means of locating these activities/objects. It is here that activity tag/object id plays an important role. Since the tag/id contains information identifying the "home" machine of the activity/object it becomes trivial to identify the machine to which the message should be forwarded. In order to simplify the implementation the AC listens at the same well known address on each machine. This eliminates the need to maintain a list of AC locations. In the event an activity/object migrates to another node, the home node would maintain a "forwarding address" for it. However, the issue of object/activity migration is not addressed in this implementation.

The use of a partitioned database necessitated the introduction of several new functions to the AC:

(19)+ Remote Activity/Object Registration List Operations - If any of the preceding commands applies to an object or activity that is not local to the machine from which the request originates, it is forwarded to the appropriate remote AC via the ac_sendwr function (see Appendix A). The distinction between local and remote objects/activities is a function of the database design and implementation as noted above. The following "remote" operations are handled by the AC:

Remote Object Registration - The AC is supplied with the id, implementor and manager ports for the object and the tag of the activity in which the object is to be registered. Provided the activity record whose key is "tag" is found, the object is added to the registration list of the object. This function is performed when a local object is required to be registered in a non-local activity.

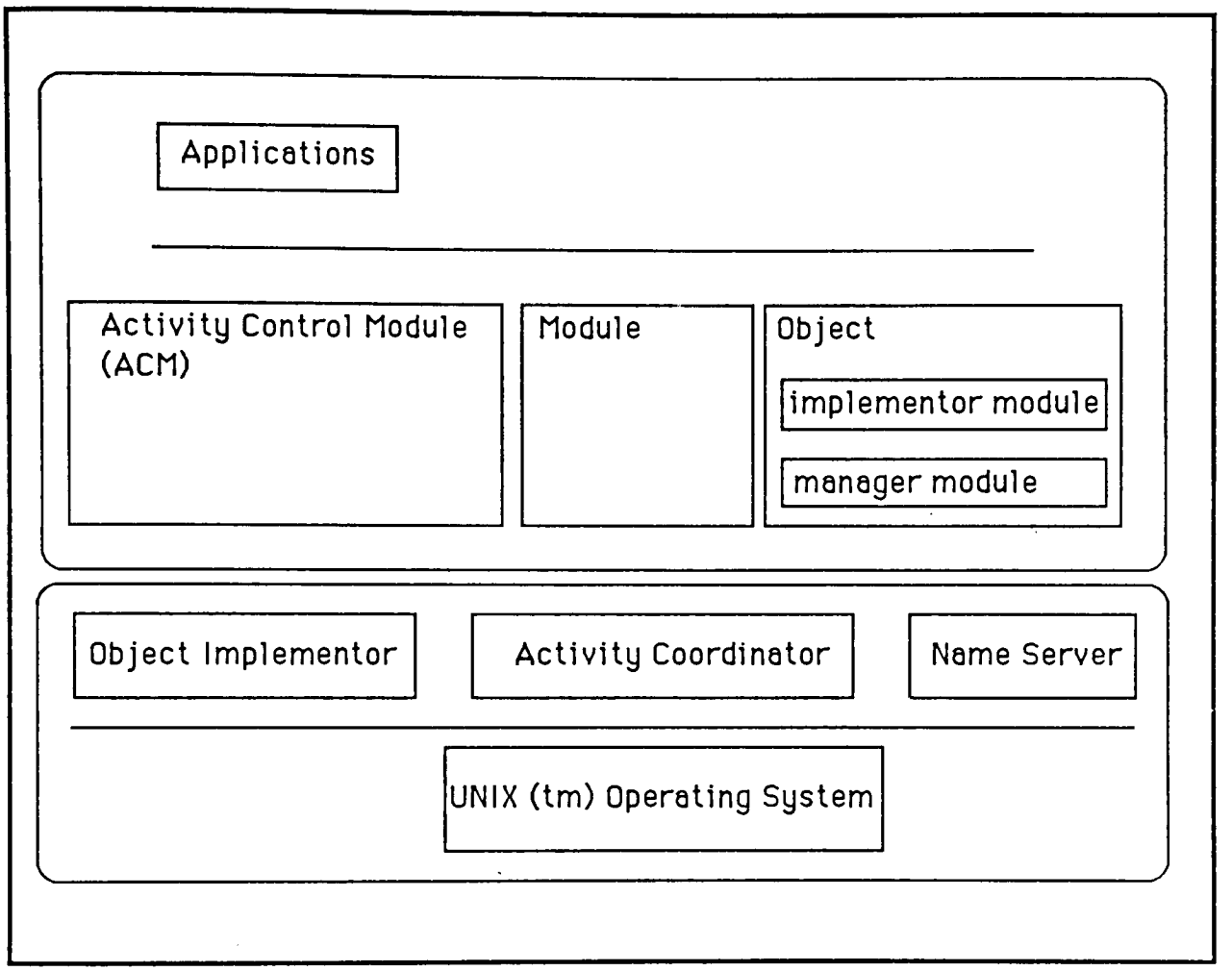
Remote Object Deregistration - Similar to Remote Object Registration, except that the object is removed from the registration list of the activity. Also, the AC is supplied only with the id and the tag, since it does not need the implementor and manager port information to deregister the object.

Remote Activity Registration - Similar to Remote Object Registration, except that it is applied to the activity registration list of an object. This function is used when a remote object is registered in a local activity.

Remote Activity Deregistration - Similar to Remote Object Deregistration, except that it applies to the activity registration list of an object.

It is clear, at this point, that the basis for certain implementation decisions regarding the Activity Coordinator have not been addressed. I will discuss these issues and provide a rationale for the various implementation decisions in Chapter 7, "Implementation Issues". In Chapter 7 I will also address issues arising out of the implementation of the simulation.

Activity System Components



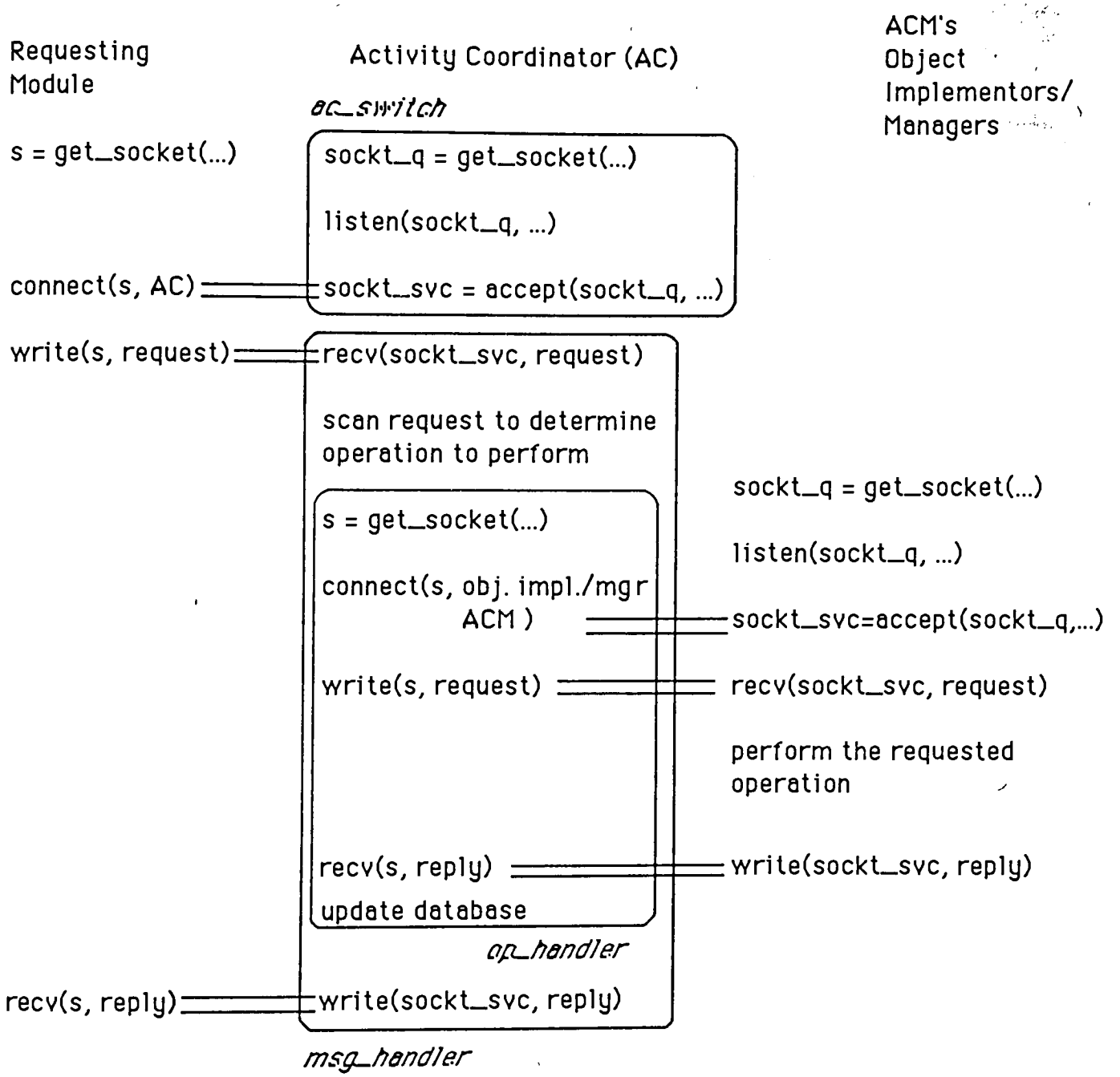


Figure 5.1
Activity System Operations - Event Sequence

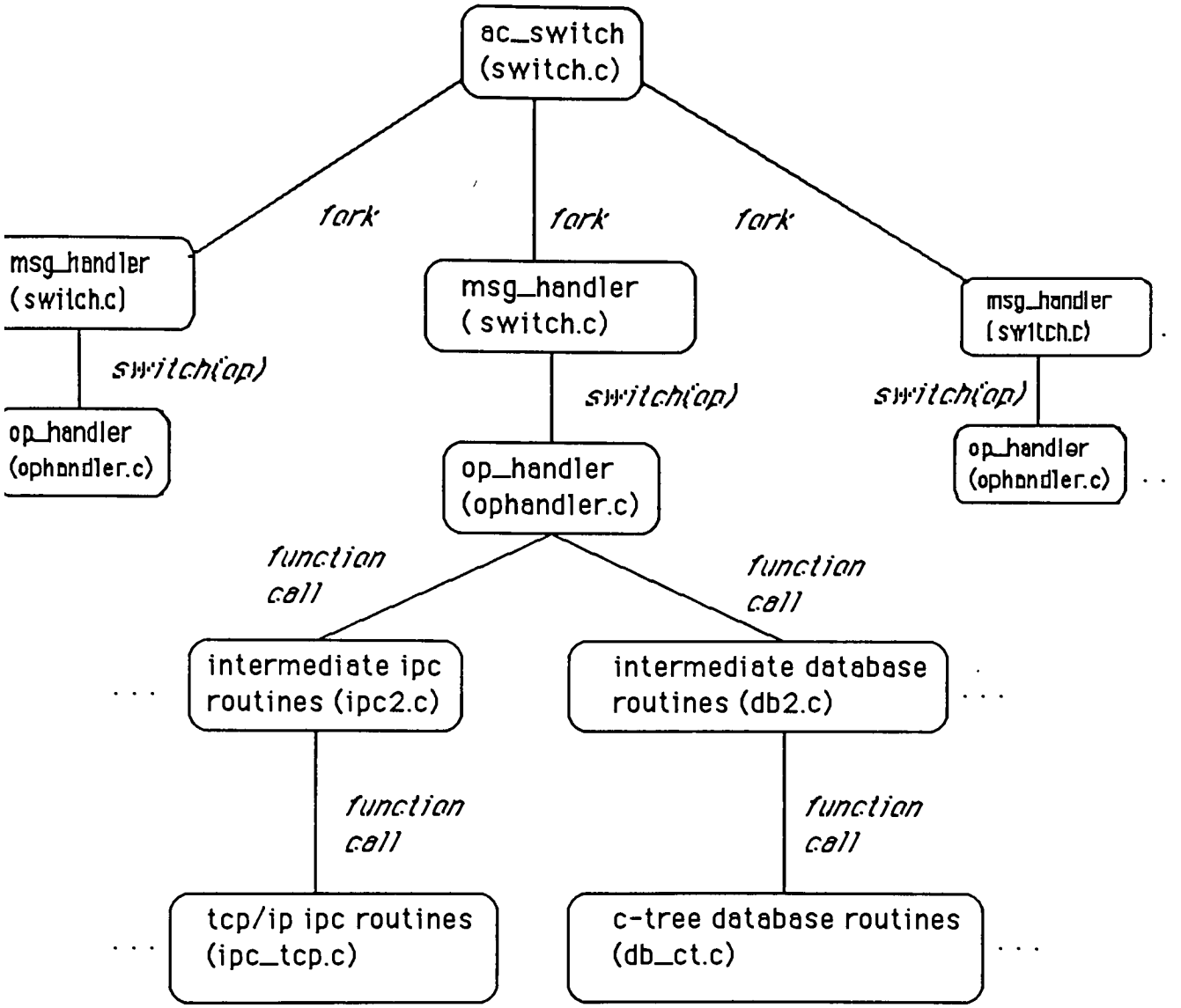


Figure 5.2
Activity Coordinator -
Functional Structure (filename)

Chapter Six

A Simulation

In order to verify the functions of the activity coordinator and to gain some experience programming in an "activity-like" environment I have implemented the following simulation: The system consists of automated assembly lines for a lithography operation. In this system, printed materials such as magazines, pamphlets, and single page documents are produced and packaged, and packaged documents are loaded onto pallets for shipping. Determination of which items are to be produced is under the control of a job scheduler. Various production jobs may have different requirements: They may use different sized boxes and/or different numbers of documents per box. Folding and binding may or may not be needed. Pallets may be packed differently for each shipment. It is up to the scheduler to initiate execution with appropriate parameters for each of the components. Once a job has been initiated, the interaction of the components within the production system controls the operation.

Figure 6.1 shows the assembly lines for the production process. The objects in the system are robots (A1,A2,B1,B2,C1,C2,D1,D2), conveyor belts (A,B,C), printing machine, folder/binder, and box sealer. The overall activity is the production of boxes containing printed materials. It is comprised of the subactivities Document Production, Packing, and Pallet Loading. Each of these works under a parent management activity: Production Management, Packing Management, and Shipping Management, respectively. Document Production may be further sub-divided into activities representing Printing, Folding/Binding, while packaging includes Box Production, Loading and Sealing. (In order to simplify the model slightly, I have assumed that multi-page documents are assembled as they come from printing. Thus, the operation need not involve any further sub-activities). Note that each of the "production" activities as well as pallet loading must be coordinated with the corresponding supply operation. These would be part of "supply" activity that share objects with the production activity. Robots A1 and A2, conveyor A, the printing machine and folder/binder belong to the document production activity. Robots B1,B2,C1,C2, conveyors B and C, and the box sealer are all components of the packaging activity. Robots B1 and B2, and conveyor B belong to box production, robot C1 and conveyor C to box loading and robot C2, conveyor C and the box sealer to box seal-

ing. Robot C2 also belongs to pallet loading, along with robots D1 and D2. Figure 6.2 shows the activity structure of the system.

There are a number of events to which the system must be able to respond, including the failure of any one of its components. Consider the failure of a conveyor belt: If conveyor belt A were to fail, the document production activity would fail. This would cause the failure of the entire activity. Packing and pallet loading would both be notified, via the AC, of the death. Box loading and sealing would then clean up the remaining items in their areas. Box production could continue, albeit at a slower rate, until such time as it needed to be suspended. Similarly, in the event that Conveyor B failed, the rest of the packing sub-activities would "clean up" their work and document production would be slowed, but not necessarily halted. The death of Robot A2 could be handled by changing production jobs to single page printings, thus removing the need for the object from the system. The failure of Robot C2 could be handled by having robot D2 load unsealed boxes onto the pallet, to be completed at a later time. This could be achieved within the context of the packing and loading activities, without involving production upstream. The death of other robots would be more serious, and would result in the failure of the activities in which they were registered. This would ultimately result in the failure of the entire production process.

Failures are not the only event that must be handled by the system. The orderly change from one printing job to another, as determined by the scheduler, can also be accomplished using activities. The scheduler sends an activity command to the root production activity to shut down the current job. The bottom up nature of the activity command execution allows the assembly lines to be cleared. When the root ACM receives the terminate notice, it returns the reply to the scheduler, which in turn initiates the new job.

Activities can also be used to synchronize operations in the absence of failures or job changes. For example, in addition to controlling actual physical operations, a module within the Box Loading activity might also monitor arrival and servicing rates. If it appears that document production and box production are not "in sync" it can notify the appropriate ACM's. Assembly and/or conveyor speeds could be adjusted, thus avoiding a failure. Similarly, if the Box Sealing ACM notices that it is falling behind, it can notify the parent activity and the sub-activities "upstream" can be told to make the necessary adjustments.

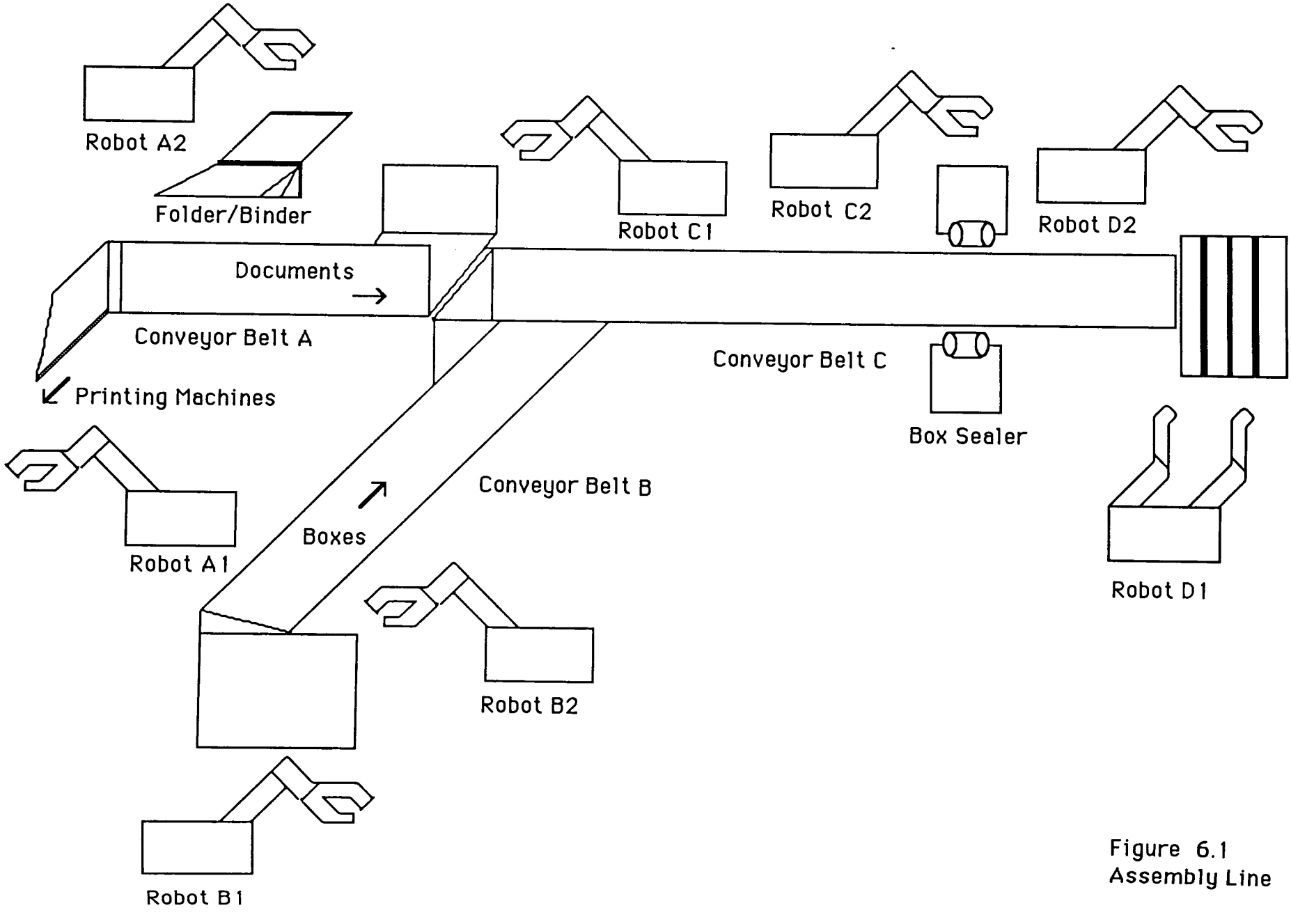


Figure 6.1
Assembly Line

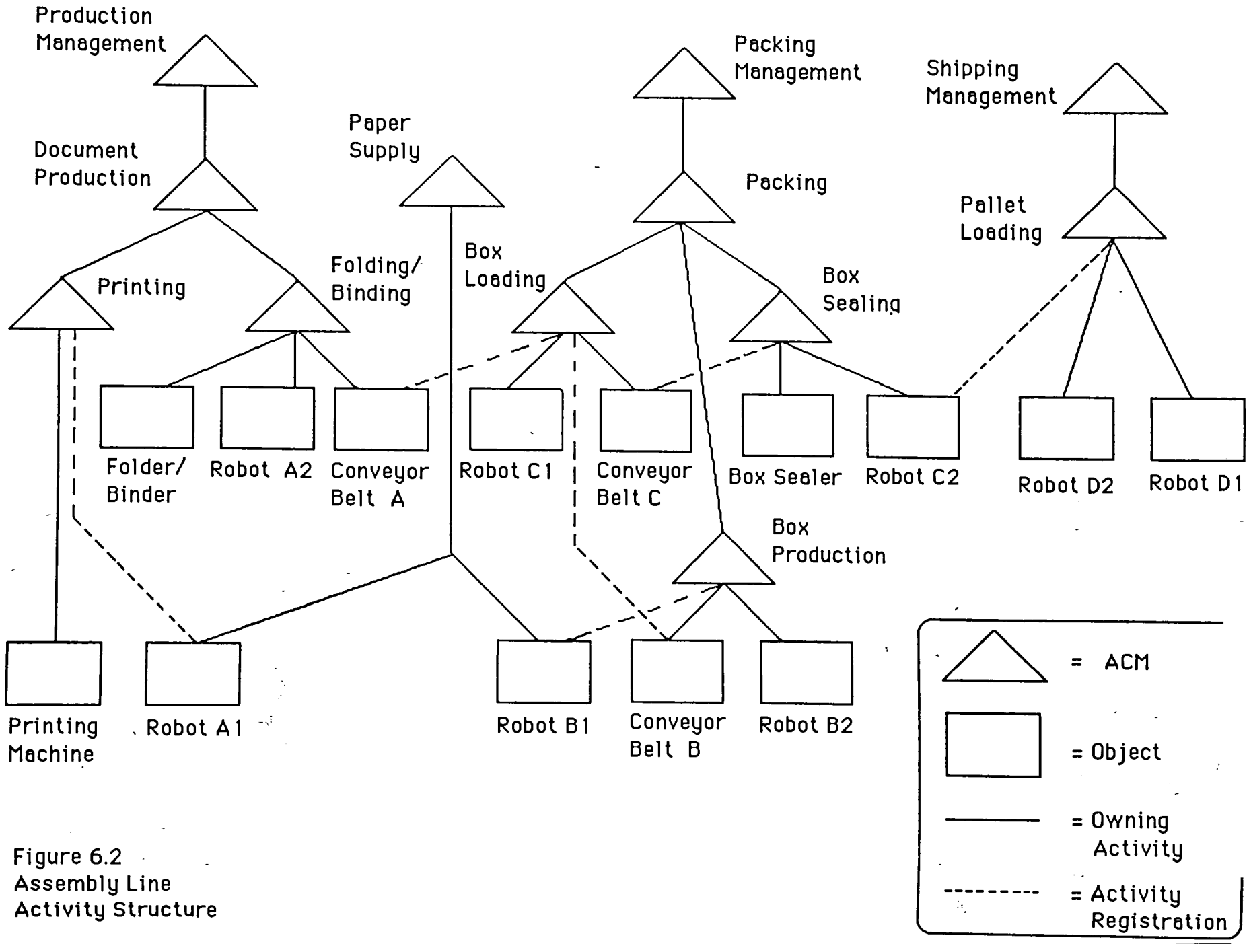


Figure 6.2
Assembly Line
Activity Structure

6.2 Simulation Design and Implementation

In order to implement the simulation, two additional servers were developed (the AC being the primary one), along with the object modules and ACM's described below (section 6.5) and a set of test programs that allow a user to request individual activity operations from a terminal (section 6.7). Both the Object Implementor and the Name Server represent simplified versions of those servers that would be found in a "real" activity system. Nevertheless, their functionality is sufficient to provide a demonstrable activity system.

6.3 Object Implementor

The Object Implementor is loosely based on the Module Implementor described in [Mayott88], but with some of the built-in fault tolerance features removed and greater parallelism in the handling of requests added. Whereas the Module Implementor was able to detect the death of modules via a separate "watcher" process associated with each one, the Object Implementor and associated test modules must simulate this feature. On the other hand, by providing a database that can be accessed in "multi-user" fashion, operations can be performed in parallel, thus freeing the server to process more requests. A further simplification of the Object Implementor arose from the fact that only those operations that involved the Activity Coordinator were implemented. In a "real" Object Implementor Connect/Disconnect operations would have been needed, but were not provided here (see below - Object Implementor Functions).

6.3.1 Modules and Processes

In order to understand the implementation of objects and ACM's in an activity-based system, the relationship between UNIX processes and Activity System modules must be addressed. As in the other systems previously discussed, the implementation of modules in the activity environment is dependent upon the underlying operating system. Particularly relevant in the UNIX environment is the the ability of a process to "fork" and "exec" other processes. A UNIX process may initiate a separate process using the "fork" system call. The child process thus created executes a copy of the original program file but within a different context from the originating process. A process may also load and execute another program file using the "exec" system call. In this case, the second process

overlays the original process. In both cases, the newly created process inherits the open file descriptors (including sockets), and process group of its originator. But, whereas in the case of a forked process, the child inherits the signal handlers of its parent, in the case of exec it does not. It is this ability to create several processes from within one program that underlies much of the design of activity system components. Under UNIX, and in particular within this implementation of an activity system, a module is roughly equivalent to an executable program file. As such, it may engender several processes when executed. In addition, since a module also encapsulates data, the definition of a module within this environment should be extended to include database files accessed by the program. When viewed from the standpoint of the object model used in the Activity System, each of the servers (Activity Coordinator, Object Implementor, and Name Server) is therefore an individual module. Each consists initially of a single UNIX process, but in the course of handling requests, each creates (forks) a number of additional processes (See sections 6.3.2, 6.4.1). As the name suggests, an Activity Control Module (ACM) is in fact a single module. On the other hand, objects, owing to the fact that they have both implementors and managers, consist of 2 modules. In this implementation of objects and ACM's, each module consists of a maximum of 2 UNIX processes, a parent "listener" process and at most 1 active child process (see sections 6.5.1, 6.5.3).

6.3.2 Object Implementor Design

The organization of code within the Object Implementor closely follows that of the AC: The main listener portion of the server, known as "obj_switch", listens at a well known address. It accepts connections from the user and forks a "msghandler" process to perform the operation. After passing the connection to the "msghandler", the "obj_switch" closes the socket and is ready to accept further requests. The "msghandler" reads and parses messages received on the socket, and performs the requested operation, via an object implementor ("obj_imp") routine. The "msghandler" then returns the result to the user on the socket that is passed to it by the "obj_switch".

6.3.3 Object Implementor Functions

The Object Implementor handles the following operations:

(1) Create an Object - the user supplies the Object Implementor with a role, a flag to indicate whether or not the object is self managing (i.e. if there is a separate manager module), the tag of the owning activity, and the pathnames of the implementor and manager modules (program files), to be "exec'd". If the object is self managing, the manager pathname will be the null string. The Implementor gets sockets for the object's implementor (and manager) port(s). The socket descriptor(s) is (are) passed as the argument(s) to the code being executed. The Object Implementor will also tell the AC to create the object, passing it the implementor (and manager) port(s) along with the role and owner tag supplied by the user. Provided the AC returns a positive reply, the Object Implementor's database is updated.

(2) Create an ACM - Similar to Create an Object; however since ACM's are not registered with the AC (they do not get id's), the AC is not requested to create an object, and the Object Implementor's database is updated automatically. In addition, the owner tag field is replaced with the owner name field, since at the time of creation of the ACM the Object Implementor does not know the tag of the activity with which the ACM is associated. The owner name can later be used in conjunction with the Name Server to find out the owner tag, if need be (see section 6.4).

(3) Destroy an Object - the user provides the Object Implementor with the id of the object to be destroyed. The Object Implementor first notifies the Activity Coordinator that the object is about to be destroyed (via a "Destroy Object" message). It then determines the UNIX process id(s) for the implementor (and manager) and sends a kill signal to it (them). Provided the AC has been notified and the kill signal(s) sent to the implementor (and manager) the object entry is removed from the Implementor's database.

(4) Remove an Object - Similar to Destroy an Object, except that the Implementor sends a warning signal to the process(es) rather than a kill signal, and a "Remove Object" message is sent to the AC in place of the "Destroy Object" message

(5) Kill an ACM - Similar to Destroy an Object, except that the user supplies the name of the owning activity, rather than the id, since this is how ACM's are known to the Object Implementor.

(6) Object Died - The implementor of the object provides notification that the object, whose id is supplied, has died. The Object Implementor in turn sends a "Destroy Object" message to the Activity Coordinator (as in (3) Destroy an Object). Provided the AC returns a positive reply, the object entry is removed from the Implementor's database.

(7) Object Removed - Like Object Died, except that the message sent to the AC is "Remove Object".

(8) ACM Failed - The Object Implementor is supplied with the name of the activity that the ACM controls. The Object Implementor finds out the activity tag from the Name Server. It then forwards the ACM Failed message, with the tag in place of the activity name, to the AC. Upon receipt of a positive reply from the AC, the ACM record is removed from the Object Implementor's database.

6.4 Name Server

One of the important aspects of the Activity System that was addressed by this implementation in a very simplistic manner is that of naming. All operations performed by the AC require either an object id or activity tag. However, outside of the module that creates an activity or object, these id's/tags are not known. This makes it virtually impossible for any other module to access the objects/activities. Furthermore, it would be advantageous from the user standpoint to be able to refer to objects and activities by name, rather than id's or tags. This is especially true of the test programs (section 6.7). Thus, while naming of activities and objects was not part of the original activity system thesis [Heliotis84], it proved to be a necessary and integral part of this implementation.

In this implementation, a very rudimentary name server was used. It simply keeps a file of id's/tags associated with a given object/activity name. Full pathnames are not used and uniqueness is not guaranteed by the server. The simulation was set up to provide unique object/activity names. A "real" name server would have to make some guarantees regarding unique names. Presumably, use of full UNIX pathnames (including the name of the host machine) would provide this. However, arbitrating between references to the same "basename" would be non-trivial. Perhaps more importantly, lookup of id's/tags for a given name should have some protections. A "capabilities list" could be attached to entries in the name server database in order to provide this protection.

6.4.1 Name Server Design

Like the Activity Coordinator and the Object Implementor, the Name Server is broken down into "listener" and "handler" sections. The main listener for the name server, the "name_switch" listens at a well known address and accepts connections from users, which are then passed on to the "namehandler". Upon startup of the server, the "name_switch" also initialises the list of remote name servers, because, unlike the Activity Coordinator and Object Implementor databases, the Name Server database is replicated across the system. This is how objects/ACM's can access objects/ACM's on remote machines.

The "namehandler" is the main operation handler for the name server. It maintains the local name database and also informs other name servers about updates. Note that no guarantees are made regarding the consistency of the various versions of the name database; i.e. the name server does not use any kind of atomic transaction scheme. The local database is updated prior to sending messages to other name servers. Messages are then sent to the other

name servers, but reply status is not processed. A remote name server is determined to be "down" if we can not send to it. This information could be relayed to a communications monitoring activity, were such an activity to be implemented. A "real" name server would want to proceed based on the status of replies. If a remote server failed to complete the transaction, the local server might then undo its update. (This is only one possible scenario; many atomic transaction schemes are possible) As a result of this simplified database scheme the result returned to the user reflects only the status of the operation on the local database.

6.4.2 Name Server Functions

The Name Server provides the following functions:

(1) Add Object Name - The user supplies the name of the object and its id. The Name Server adds the name to the local database and forwards the message to all other name servers in its list.

(2) Remove Object Name - The user supplies the name of the object to be removed. The Name Server removes the record from its local database and forwards the message to all other name servers on its list.

(3) Get an Object's ID - The user supplies the name of the object whose id is requested. If the name is found in the local database, the id is returned. No polling of other name servers takes place if the name is not found.

(4) Get an Object's Name - Like Get Object ID except that an id is supplied by the user and the name of the object is returned in reply.

Each of the Activity operations are identical to the corresponding Object operations except that tags take the place of id's where applicable.

(5) Add an Activity's Name

(6) Remove an Activity's Name

(7) Get an Activity's Tag

(8) Get an Activity's Name

6.5. Simulation: Objects and ACM's

Since simulation of all objects and ACM's in the system is beyond the scope of this project, I have focused on those parts of the production activity that are within one step of Robot C1. Furthermore, constraints on the number of UNIX processes that may be active for a single user have limited the number of objects and ACM's in the simulation that can be implemented on a given machine. The following object modules are simulated:

(1) Implementors/Managers for Robots A2, C1, and C2.

(2) Implementors/Managers for Conveyors A, and C.

In addition, there are ACM's for the following activities:

(1) Document Production, including subactivity Folding/Binding,

(2) Packing, including subactivities Box Loading, and Box Sealing.

(3) Pallet Loading

(4) The respective Managers for Document Production, Packing and Pallet Loading.

These objects/ACM's will be distributed between machines as indicated in figure 6.3. Note that the distribution corresponds to the subactivities Document Production, Packing, and Pallet Loading. The following sections describe the design and implementation of objects and ACM's.

6.5.1 ACM Design

Each of the ACM's consists of 2 component UNIX processes. The parent is the "listener" process; it handles communication for the activity that the ACM controls. Upon receipt of an "ACM_STARTUP" message (see below), the listener forks a child process. This child process handles activity initialization functions including the creation of sub-activities. In this implementation, the child process then exits. The alternative would have been to have it loop forever, awaiting signals from the listener based on further communications that the listener receives, and proceeding based upon these signals. This alternative design was implemented for other system objects (see section 6.5.3 - Object Design). For the most part, each of the ACM's are identical, the primary difference being which sub-activities they initiate.

6.5.2 ACM Functions

The ACM handles the following functions:

(1) ACM_STARTUP - The user supplies the tag of the activity that the ACM controls. The listener spawns a manager process as its child. The manager process uses the activity tag information in its initiation of other activities and/or objects.

(2) ACT_SUSPEND - The user supplies the tag of the activity on whose behalf work is to be suspended. In this simplified implementation, the listener does nothing except acknowledge receipt of the request, since there is no child process to signal. Execution of the listener process continues.

(3) ACT_FREE - Similar to ACT_SUSPEND. Again, this action takes placed independently of the tag provided. The listener simply replies to the request and continues execution.

(4) ACT_TERMINATE - Similar to ACT_SUSPEND except that execution of the listener is terminated.

(5) OBJ_DIED - The user supplies the id of the object that has died. Actions taken by the ACM for this command vary, but minimally the listener responds to acknowledge receipt of the message. The listener may fork a new child (manager) process to handle the operation.

(5) ACM_TERMINATED - This message is received by the ACM upon termination of the child activity whose tag appears in the message. Like the OBJ_DIED notice, actions taken by the ACM for this command vary, but minimally the listener responds to acknowledge receipt of the notice.

(6) User-Defined Activity Commands - As for the previous 2 commands, actions taken by the ACM for these commands may vary, but minimally the listener responds to acknowledge receipt of the message.

6.5.3 Object Design

Each of the robots and conveyor belts was implemented with separate modules for the implementor and the manager. Like the ACM's, each of these modules consists of 2 component UNIX processes: In both the implementor and manager, the parent process is a "listener" that handles communication for the object. It is also responsible for sending the appropriate signals to the child process based on this communication (see below). In the case of the

implementor, the child process represents the simulated managed machine. In the case of the manager, the child process takes care of initial activity related functions, such as registering the object in activities (other than its owner) and creating additional objects. The principle difference between the child process in the object manager and that of the ACM is that it does not exit after initialization, but rather continues execution. For both object implementor and manager execution of the child process begins only upon receipt of the OBJ_STARTUP command. The child process is designed to complete execution of it initialization at startup. After this, it performs work in accordance with certain flags. These flag values are set once at initialization, and thereafter by the signal handlers within the child process. Like the ACM's, the execution of each object is nearly identical, the primary differences being in the operations of the machine and in the activity functions performed by the managers.

6.5.4 Object Signal Handling

Note that for both the manager and machine processes the work of the child is asynchronous to that of its listener. Therefore, changes in the state of the object (which are made in response to messages sent to the listener) are handled through signals. Signals sent to the child by the parent (the listener) trigger the appropriate signal handler, which sets the corresponding flags. The functions performed by the child are controlled by the value of these flags. It was originally intended that the functions be part of the signal handler itself, but this turned out to be impossible within the current UNIX System V implementation (see section 7.3.5). It was not the goal of this thesis to develop objects that exhibited "realistic" behavior, but simply to respond to commands as required to test the Activity Coordinator. One area in which this design differed from that of a "real" activity system was that of establishing an activity context for the processing of signals (see section 7.3.4). Nevertheless, it was essential that the signal mechanism function properly, in order to demonstrate "reasonable" responses to activity commands. As a result, the objects support only enough signals to handle the functions described below.

6.5.5 Object Functions

Object Implementor/Manager modules perform the following functions:

(1) OBJ_STARTUP - The user supplies the id of the object being started. If the message is directed to the object manager, the listener spawns a manager process as its child. The manager process uses the id when it registers (deregisters) the object in (from) activities. If the message is directed to the object implementor, the listener spawns a simulated machine process as its child. In this case, the id is used when the object receives a DESTROY_OBJ or REMOVE_OBJ message (see below).

(2) REGISTER_OBJ - This message is sent to the object's implementor/manager by the AC at the time of the object's creation and also when the object is to be registered in an activity other than its owner. The tag of the activity in which the object is to register is provided in the message. This function is intended to provide the capability for objects to execute within different activity contexts. However, at present this function is implemented only insofar as the listener replies to the request. No actual change in the execution of the object is made.

(3) DEREGISTER_OBJ - Like REGISTER_OBJ, this message is intended to establish activity context in which the object executes. Likewise, this function is not implemented, except to the extent that the listener will reply to the request.

(4) ACT_SUSPEND - The user supplies the tag of the activity on whose behalf work is to be suspended. The listener sends the appropriate signal to the child process, independent of the activity tag provided. (This was done to simplify object implementation.) The signal handler within the child process sets the corresponding flag and execution of the child process is suspended. Note however, that this command does not take effect until after initialization steps begun under "OBJ_STARTUP" have been completed. (Again, this was done to simplify object implementation.) Execution of the parent ("listener") process continues.

(5) ACT_FREE - Similar to ACT_SUSPEND, except that execution of the child process is resumed. Again, this action takes placed independently of the tag provided.

(6) ACT_TERMINATE - Similar to ACT_SUSPEND except that execution of both the parent and child are terminated.

Note that for each of the above activity commands, the operation takes place independently of the activity on whose behalf the command is invoked. This was done to simplify the implementation and was due to the difficulty of establishing an activity context for object execution. In a "real" activity system, an activity context mechanism would be a necessary part of an object implementation (see section 7.3.4).

(7) DESTROY_OBJ - Results in the immediate death of the object (without any cleanup of existing work). This message was added in order to provide for simulation of object failures; it would not necessarily be present in a "real" activity system. The listener first acknowledges receipt of the message. If the recipient of the message is the implementor of the object (as opposed to the Object Implementor), the listener notifies the Object Implementor that the object has died. (Note, this is in contrast to having an object "watcher" that observes the death of the object and performs the notification.) The listener then sends a "terminate" signal to the child process and exits. An alternative would have been to have the listener wait for a DEREGISTER_OBJ message from the AC (as part of the object invalidation process). However I felt this scenario to be more realistic in the sense that if the object dies, chances are reasonably good that its listener will too.

(8) REMOVE_OBJ - Similar to DESTROY_OBJ except that the child is sent a "warning" signal instead of a "terminate" signal, which allows it to clean up any pending work before it exits. In addition, the listener waits for the DEREGISTER_OBJ message from the AC before it exits. Unlike the DESTROY_OBJ command, this operation would be implemented in a "real" activity system.

6.6 System Startup and Operation

The startup of the system is initiated by the program "simstart" This program handles creation of the default ACM and the root activity on a given machine. Startup of the system is coordinated so that events that are required to be ordered take place in the correct sequence. Typically, this means that parent ACM's are responsible for the creation of sub-activities and objects that they own. Beyond this, however, actions of several system components take place concurrently. The overall relationship of events is shown in figure 6.4

The following section describes what each of the ACM's, robots, and conveyor belts does from a strictly functional standpoint. Note that only those that were implemented in the simulation are included.

ACM Packing Manager - This is the root activity for Packing and accordingly it starts up the Packing ACM and requests creation of the activity. In a "real" implementation this ACM would monitor production rates and send control messages to its subactivities accordingly.

ACM Packing - is created and owned by the Packing Manager. It requests the creation and initiates the startup of sub-activities Box Loading and Box Sealing.

ACM Box Loading - requests the creation and initiates the startup of objects Conveyor Belt C and Robot C1.

Conveyor Belt C - is created in the context of activity Box Loading. It registers in activity Box Sealing and then begins operation.

Robot C1 - is created with Box Loading as its owner. It neither registers in any further activities, nor initiates the start of any other objects. It begins its work of loading documents into boxes immediately upon receipt of the OBJ_STARTUP command.

ACM Box Sealing - requests the creation and initiates the startup of Pallet Loading with Shipping Management as its parent activity. This tests the capability of the AC to create an activity whose parent is located on a remote node.

ACM Production Manager - This is the root activity for Document Production Like the Packing Manager it starts up the corresponding ACM and requests creation of the activity.

ACM Document Production - is created and owned by the Production Manager. It requests the creation and initiates the startup of sub-activity Fold/Bind.

ACM Fold/Bind - requests the creation and initiates the startup of objects Conveyor Belt A and Robot A2.

Conveyor Belt A - is created in the context of activity Fold/Bind. It registers in activity Box Loading and then begins operation. Note that this tests the capability of the AC to register an object in an activity on a remote node.

Robot A2 - is created with Fold/Bind as its owner. It is identical to Robot C1, except that instead of loading documents into boxes, Robot A2 folds and binds documents.

ACM Shipping Manager - This is the root activity for Pallet Loading. Unlike the corresponding managers for Document Production and Packing, the Shipping Manager does not initiate the creation of the sub-activity Pallet Loading. In a "real" system, the Shipping Manager would presumably have other sub-activities that it did create, but in this system it exists only to act as the parent for Pallet Loading.

ACM Pallet Loading - requests the creation and initiates the startup of Robot C2 with Box Sealing as its owner. This tests the capability of the AC to create an object whose owner is on a remote node.

Robot C2 - is created with Box Sealing as its owner. It registers in activity Pallet Loading and then begins operation.

6.7 Test Tool Programs

In order to facilitate testing of the Activity System, and in particular the Activity Coordinator, I have provided ;et of test programs that allow a user to request activity operations from a terminal connected anywhere in the sys- m independently of the pre-defined operations for the objects and ACM's created in the simulation; i.e. one does ıt have to be on the machine that is the "home" machine of the operation's object/activity. The only stipulation is .at the local Name Server database know of the object/activity involved. Limitations with respect to the Name ;rver database have previously been discussed. These test tools are invoked as UNIX shell commands (i.e. they are :ecutable program files) and consist of the following:

(1) register "object name" "activity name" - The program looks up the id of the object and the tag of the activity via the Name Server. It then sends a "REGISTER_OBJ" message to the Activity Coordinator.

(2) deregister "object name" "activity name" Same as "register" except that the message sent to the Activity Coordinator is "DEREGISTER_OBJ".

(3) destroy "object name" - The program looks up the id of the object via the Name Server. It then looks up the implementor (and manager) port(s) via the Activity Coordinator using the id. A "DESTROY_OBJ" message is then sent to the implementor (and manager) port(s) of the object.

(4) remove "object name" - Same as "destroy", except that the message sent to the object is "REMOVE_OBJ".

(5) killacm "activity name" - The program looks up the tag of the activity via the Name Server. A "KILL_ACM" message is then sent to the Object Implementor.

(6)suspend "activity name" - The program looks up the tag of the activity via the Name Server. The Activity Coordinator is then requested to send the activity command "ACT_SUSPEND" to the activity tree rooted at "tag".

(7)resume "activity name" - Same as "suspend", except that the activity command is "ACT_FREE"

(8)terminate "activity name" - Same as "suspend", except that the activity command is "ACT_TERMINATE"

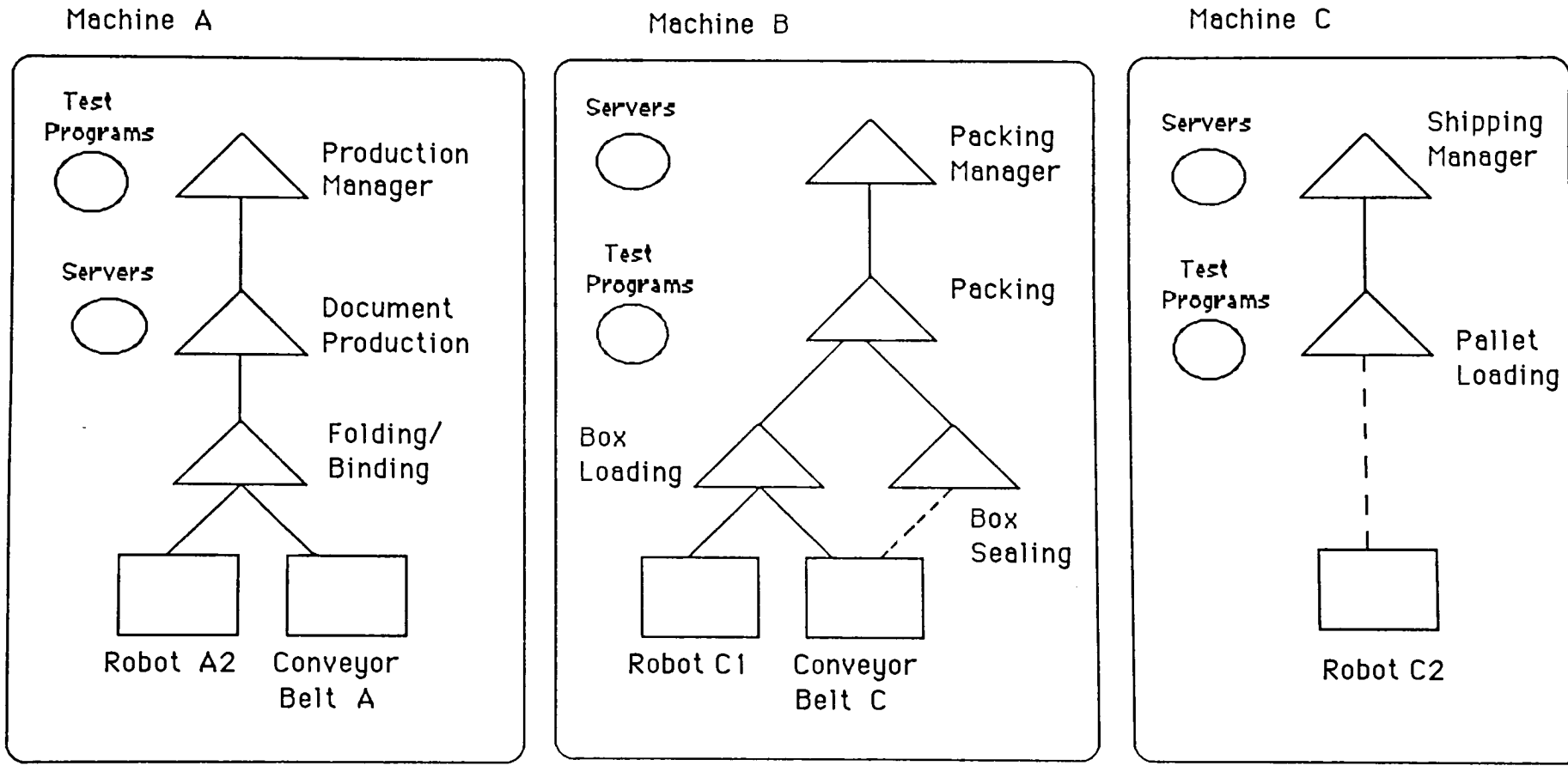


Figure 6.3
Module Distribution

↓	Packing Manager	Packing	Box Loading	Conveyor Belt C	Robot C1	Box Sealing
t1	Create Packing					
t2		Create Box Loading				
t3		Create Box Sealing	Create Conveyor C			
t4			Create Robot C1	Register in Box Sealing		Create Pallet Load
t5				Work	Work	

	Production Manager	Document Production	Folding/ Binding	Conveyor Belt A	Robot A2	
t1	Create Doc. Prod.					
t2		Create Fold/Bind				
t3			Create Conveyor A			
t4			Create Robot A2	Register in Box Loading		
t5				Work	Work	

	Shipping Manager	Pallet Loading	Robot C2			
t1 ⋮ t4						
t5		Create Robot C2				
t6			Register in Pallet Loading			
t7			Work			

Figure 6.4 - Simulation Event Sequence

Chapter Seven

Implementation Issues

Issues in the implementation of the Activity Coordinator and the rest of the activity system can be broken down into 3 categories: database issues, inter-process communication issues, and simulation issues. This chapter will examine each of these in turn.

7.1 Database Issues

A number of issues in the development of the Activity Coordinator grew out of the design and implementation of the database. These include use of primary vs. secondary storage, individual file and overall database organization, the mechanism to be used to enable concurrent access by multiple processes, and finally, database distribution.

7.1.1 Shared Files vs. Shared Memory

One of the original design goals was to make the Activity Coordinator portable between System V and Berkeley 4.2 UNIX. With this in mind, the use of shared files, rather than shared memory, for implementing the database was mandated; first, in order to permit access by multiple "ophandler" processes, and second, for reliability. Given this requirement, it became especially desirable to limit the number of disk accesses required for most operations. As we shall see, this particular goal had an important impact on a other decisions regarding the database design.

7.1.2 File Organization

Of immediate concern was the fact that a flat file organization would clearly not suffice for any kind of reasonable storage and retrieval requirements. On the other hand, supporting the overhead of something like a relational database is clearly unwarranted for the simple types of operations required by the AC. A B-tree indexed file system provides an efficient way to manage record storage and retrieval without incurring undue overhead. The basic rules for a B-tree of "order" n are as follows [Wirth76]:

(1) Every node must contain at most 2n keys.

(2) Every node except the root must contain at least n keys.

(3) Every node except leaves must have at least m+1 children, where m is the number of keys in the node.

(4) All leaves must be at the same level.

Some trees may relax rule (2) for leaves to allow new leaf additions with less disruption of the rest of the tree. Some may carry upper node values down the tree so that all values may be found in the leaves and only leaves actually have pointers to the data file. There are, in fact, many B-tree variants, but a discussion of these is beyond the scope of this thesis. For a comparison of some B-tree algorithms see [Fishbeck87]. Given a B-tree indexed file structure that implements the above requirements, locating a 10-byte key in a file of a million requires at most 5 disk accesses [Lewis87]. While this number of disk accesses is unacceptable for most real-time applications, this is a worst case scenario, and I would certainly not envision the AC database being this large.

Given the fact that writing a reasonably complete, efficient B-Tree Library is a non-trivial task and that there are a considerable number of commercially available packages for this purpose, it made little sense to develop my own version from scratch.

7.1.3 C-Tree

Purely practical considerations led to the use of Faircom's "C-Tree" file manager package; namely it was available on the machine upon which I did most of the system development. This was fortunate because C-Tree, in addition to being a very flexible and thus powerful package, also comes with source code; should one really want to modify the system, it can be done. Some of the features of C-Tree that were particularly useful in this implementation include:

(1) Parameter Files - A single file can be created to describe the necessary characteristics of the data and index files needed by an application (essentially, a data dictionary file). A set of functions are provided that use this file for creating, opening, and closing all the data and index files simultaneously.

(2) Flexible Key Types - Binary, floating-point and composite key fields are supported. Of particular use in this system, compression of duplicate leading and trailing characters in strings is also provided.

(3) System Configuration - memory size for I/O cache, index node size, file extension size, and automatic write-through to disk are all configurable. One could potentially fine-tune the configuration for the AC database.

(4) Deadlock Free Index Record Locking - The C-Tree index node locking protocol is guaranteed to be deadlock free, by virtue of its use of the System V "lockf" system call in blocking mode. The lock request allows the process to "sleep" if the lock is not immediately available. On the other hand, data record locks are not guaranteed to be deadlock free. Therefore, the data record lock routine returns instead of sleeping when a lock is denied because of a competing lock. The application program must deal with the appropriate action if a lock is denied to a data record (see section 7.1.5). Of course, the C-tree data record locking routines could be modified to block on data record locks as for index nodes.

(5) Concurrency Control - The particular variant of B-tree implemented in C-Tree enables simultaneous updates and searches of the same index node with minimal locking requirements. Additionally, it permits nodes to be split even while other users are accessing the same nodes.

7.1.4 Database Design and Organization - Data Structures

Within the context of C-Tree there were additional considerations regarding database organization. This was particularly relevant in the area of object and activity registration lists, and the activity tree structure. Had the data structures been linked-lists and trees within main memory this would have been very straightforward. However, the use of shared files brought serious concerns with respect to efficiency; again, it was of primary concern to minimize disk accesses for most operations.

For object (activity) registration lists I considered the following alternatives:

(1) Arrays of object ids (activity tags) within each activity (object) record

(2) A separate file for each activity/object with individual records for each list element

(3) A file of object/activity pairs with indices maintained for both activity tags and object ids.

(4) Variable length strings of ids (tags) within variable length records. This would involve treating the list as a character string and using string functions to add/delete list elements.

The current implementation uses the array option (1). The primary advantage of this method is that it speeds storage and retrieval of list elements. Traversing the list would have required multiple disk accesses for options (2) and (3). Not only would this have been slower, but also would have increased the chances of a transaction failing to obtain locks. Arrays were selected over option (4) primarily for 2 reasons: 1. Again, number of disk accesses (c-tree requires an additional read to obtain the variable length portion of a record), and 2. In terms of the string operations required to manipulate the lists, the current implementation requires somewhat less overhead. The main disadvantage of this design is that it reduces flexibility (since the list has a fixed size limit).

The alternatives considered for the activity tree were similar to those for the object registration lists:

(1) Store child activities as an array of tags (as for registration lists)

(2) Have a separate file for each activity with records for sub-activities

(3) Have a "sub-activities" file with entries indexed by parent activity

(4) Store tags of first child and right sibling as pointers, and traverse the tree by reading the corresponding records from the activity file.

Options (2) and (3) were discarded for the same reasons as above. Option (1) presented the best choice from the standpoint of disk reads. However, one additional factor was significant in the activity tree design decision. One of the main operations that uses the activity tree is that of sending an activity command. This requires that the structure chosen for the activity tree must readily support recursion. While it was possible to get the desired results using option (1), option (4) was clearly better suited to this operation. Furthermore, since traversing the tree inevitably requires multiple disk reads, option (4) was chosen.

7.1.5 Database Concurrency Control

Another important issue in the implementation of the database was that of providing concurrent access by multiple processes, while avoiding deadlock. To this end, all database updates were performed using the following algorithm: A record is read without obtaining a lock on the data file. The modification is made to a "new" buffer area. A lock is then obtained and the "new" record re-written to the database. If the process is not able to obtain a lock, it sleeps and tries again. If there has been a modification to the record by another process in the time between the original read and the attempt to write the record back to the database it re-reads the record and tries again. If the process is unable to perform the update within a maximum number of tries, it gives up and the update fails. Note that this algorithm is not "starvation" free since a process may run out of "tries" before it is able to make the update, but since the chances of repeated update failure are small and there is a "conflict resolution" mechanism built in, it is at least "reasonable".

Typically, updates to an activity occur within the context of one or two modules, and are thus not likely to create massive conflicts. Indeed, the motivation for concurrent access to the database was to allow non-conflicting requests (those referring to disjoint sets of activities), to proceed in parallel. It was not designed to handle large numbers of conflicting requests. Again, I consider the former case to be far more likely in the context of activities. Note that the integrity of the database is assured with respect to conflicting requests, but that no further consistency guarantees are made regarding atomicity of updates (see section 8.2.1).

7.1.6 Partitioned Vs. Replicated Database

Another issue addressed in the implementation of the AC database was whether it should be partitioned or replicated. The use of a partioned database in this implementation was motivated by several factors. First, there is reduced communication overhead associated with this type of database. This advantage is particularly apparent when the majority of the operations can be performed locally. The hierarchical structuring of activities, sub-activities, and objects lends creedence to the notion that this will indeed be the case; i.e that many activity-based applications will be organized as in the simulation, with activities grouped together logically on a particular machine. Second, although the size of the database was not a primary consideration, use of a partitioned database results in

smaller files and thus better access times. Third, the use of a partitioned database was more in keeping with one of the original goals of the AC distribution; namely, that of maintaining local autonomy of each AC server. It was presumably never a goal of the original AC design that any one AC server have a complete global view of the entire activity system. Rather, "each piece remembers locally enough of the activity tree to handle failures [and other events] inside or outside its own machine that affect its resident activities" [Heliotis84].

On the other hand, increased availability and fault-tolerance could have been provided via a replicated database. (Note that the current implementation is not at all fault-tolerant). In this case, the issue becomes one of maintaining multiple versions. It is my belief that, within the current development environment, maintaining consistency of multiple versions would create sufficiently high overhead as to be impractical, particularly for real-time applications. In addition to the communication overhead incurred, an underlying atomic transaction scheme would have been needed in order to maintain consistency of the databases across machines. This is not to suggest that a replicated database could not or should not be implemented as part of a "real" activity system. In fact, since one of the principle goals of an activity based distributed system is to provide greater reliability, the underlying servers (particularly the AC) should be made as fault-tolerant as possible. This would include at least implementation of a replicated database.

7.2 Inter-process Communication Issues

A number of issues in the implementation of the Activity Coordinator also arose out of the underlying inter-process communication mechanism. These include the use of the Wollongong Group's Socket compatability Library ([WIN86]) and particularly the attendant limitations, the handling of emergency notices, and the use of timers to prevent communications deadlock.

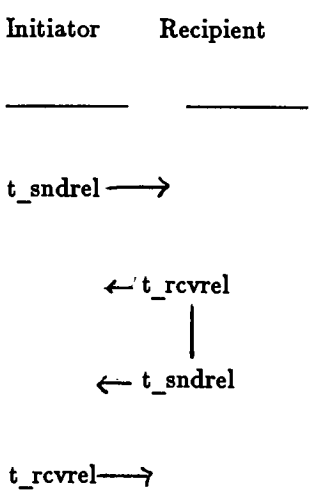
7.2.1 Sockets vs. Transport Level Interface

The use of the WIN/3B Socket Compatability Library as opposed to System V Transport Level Interface (TLI) was one instance where an attempt to make the implementation compatible with 4.2 BSD UNIX(tm) restricted the capabilities of the system. However, having originally developed the communication routines using sockets, and given the time constraints, I felt that it was necessary to keep the initial implementation and to find "workarounds" for the deficiencies in the socket library. Potential advantages of a TLI-based implementation are noted below:

The capability to manage socket options is not implemented in the socket library. This means that local address reuse on binds can not be enabled, sockets can not be set to linger on close if unsent messages are queued, and perhaps most importantly, sockets can not be set to keep connections alive beyond the system default time limit. Each of these options is provided under TLI. More importantly, TLI provides additional functions for transport level communication management.

The Transport Layer Interface provides functions for additional handshaking between clients and servers. This in turn means better control over event sequence and the ability to detect when the protocol between client and server has been disrupted. The function t_sndrel initiates the "orderly release" of a connection, and is used to indicate that the sender has no more data to send. A call to t_sndrel by the client generates an orderly release indication to the server. The client then waits, using t_rcvrel, for the server to send an "orderly release" of its own. Of course, the roles of client and server can be reversed in this scenario.

The protocol proceeds as shown below:



Note that if either side fails to receive the orderly release indication it will know that there is a problem with the connection.

TLI provides functions that are particularly useful in diagnosing such problems, as well as in failure recovery. Under TLI one is able to read the current state of a transport endpoint via t_getstate. This provides information concerning the next expected event. One is also able to look at the current event on a transport endpoint via t_look. This is especially useful in determining the cause of transport layer communication failures, since one may observe the occurence of incompatible events and treat the problem accordingly. For example, both sockets and TLI have the same deficiency with respect to sending messages; namely, if the connection is broken before data transmission is completed, data will be lost without any notification to the sender. However, using TLI, the listener is able to detect this situation, by monitoring the incomming side of the connection for an unexpected "disconnect" event. Moreover, once the disconnect is received, the user may determine the reason using t_rcvdis.

Using these functions, most transport level communication problems can ultimately be detected and fixed. Should a server be unable to correct a problem within itself, there is one last recourse open to it under TLI, that is not available with sockets. If it becomes absolutely necessary, a server may unbind its designated address using t_unbind. It may then try to re-bind the address and continue processing. Of course there is no guarantee that the

server will be able to re-bind its address and continue functioning as before. However, TLI gives us one more possible option for dealing with a communication failure. Using sockets, the server process would have to be killed and re-started. In short, the TLI interface could have been used to implement a much more robust Activity Coordinator, as well as other simulation system components.

7.2.2 Emergency Notices

When an object fails, the Object Implementor and the Activity Coordinator receive an "emergency notice" regarding the death. It would be desirable to have these notices receive attention with higher priority than "normal" operation requests. Under the current implementation, however, these messages arrive in the normal stream of communication; i.e the process sending the notice is queued along with those initiating other kinds of requests, awaiting connection to the server. One alternative would be to send and receive these messages using the "out of band data" facility ([WIN86], send(3W), recv(3W)). This would require data sent by a user prior to the receipt of the higher priority message to be buffered for later servicing, while maintaining the connection to these clients. As noted in the preceding section, this may not be possible due to the inability to manage socket options. Furthermore, the fact that these requests must be read anyway means that we have not really gained much by the use of "out of band" message services. Of course, prior requests could simply be flushed. But this means that all users would then be responsible for retrying their requests.

A better alternative would be to associate a second socket with both the Activity Coordinator and Object Implementor "switchboards". The "switchboard" could then do a "select" before accepting connections to see which sockets had pending requests. The socket that was reserved for emergency notices would receive priority service over the socket for "normal" messages. Of course, this implies that there is some control over who has access to the emergency notice socket. Under the simplified implementation of objects and of the Object Implementor, this control is not practical since it is the object itself that sends notice of its own death. However, had "watchers" been implemented as in [Mayott88], access to the emergency sockets could be limited to these modules. Since creation of "watchers" would be under the direct control of the Object Implementor, so would access to these special sockets.

7.2.3 Timers and Communication Control

As noted in the functional description of the Activity System provided in chapter 4, one of the requirements for messages sent in activity system operations is that a response be received in a "reasonable" amount of time or the operation is assumed to have failed. In this implementation, timeout values for each operation were set to a default REPLY_TIMEOUT, that is contained in each server's respective include files, and that is therefore fixed at compile time. A better method would have been to enable the user to include the timeout as part of the message that invokes the operation. The implementation of timer controls in the current system is itself somewhat deficient, owing to a problem with the System V socket compatibility library and the kernel implementation of real-time timers. Had the system been implemented under 4.2 BSD UNIX(tm), this limitation would not have been present. It is a documented feature of the System V Socket Compatibility Library that one may "select" in order to receive from one or more socket descriptors and pass as one of the arguments a pointer to a timeout structure. This timeout structure enables the time to be defined with microsecond granularity. However, attempts to use this feature produced kernel panic errors. Consequently, the current implementation uses a somewhat inefficient technique, and one that enables timeout granularity only in seconds. In order to receive from a single socket, the process uses a blocking "receive", together with the "alarm" function and a simple alarm signal handler. In order to receive from multiple sockets, the process uses "select" but with the timeout pointer set to null, thus effecting a blocking read, together with the same alarm signal arrangement. This deficiency in the timer control over communications did not pose a problem for the simulation system. However, should this system be used for real-time applications, it would have to be remedied.

7.3 Simulation Issues

Finally, there were several issues unrelated to database implementation or inter-process communication that came up in the development of the simulation. These are examined in the following sections.

7.3.1 Processes

Under the current System V implementation where the simulation was developed, the number of processes alloted to any single user is 24. This number of processes is inadequate for a large scale simulation. Since objects may

have both implementors and managers, and (under the current implementation) each of these is composed of 2 component processes, each object may have 4 processes associated with it. Typically, ACM's will have up to 2 associated process. The number of processes associated with each of the servers will vary depending on the operations required, but typically is around 1 or 2. This meant that if, for example, there were 4 ACM's in addition to the default ACM, an AC, Object Implementor, and Name Server on a given machine there could be no more than 3 objects simulated on that machine. Allocating more processes to the Activity System could be accomplished using system administration functions associated with System V UNIX(tm).

7.3.2 Signals

A number of implementation issues within the development of objects and the Object Implementor arose around signals and signal handlers. In most instances, there were advantages to the facilities provided under Berkeley 4.2 UNIX(tm) over those available under System V UNIX(tm).

One place where a difference between the 2 versions of UNIX(tm) arose was in the signal processing done by objects. In order to Suspend/Free a module the System V version used SIGUSR1 and SIGUSR2 and (trivial) handler functions to simulate the effects of SIGSTOP and SIGCONT in 4.2 BSD.

7.3.3 Signals and Process Groups

A second difference, and one which posed a problem in the System V version, arose in the use of process groups. Berkeley 4.2 UNIX(tm) provides separate "kill" and "killpg" system calls, the latter being used to send a signal to a process group as opposed to a single process. On the other hand, System V has only "kill". It was my experience that kill did not function for a process group when it had been established within the context of the object's parent "listener" process, and when the kill originated from a process outside of the process group (in this case the Object Implementor). Instead, kill worked only for the parent of the group. This eliminated the possibility of destroying/removing objects via a message to the Object Implementor. Instead, the message had to be directed to the implementor (and manager) of the object itself.

7.3.4 Signals and Activity Context

More significant than differences in the 2 versions of UNIX(tm) was the fact that, in the simplified simulation implementation, the object implementors/managers were not able to determine the activity context that a code block was running in. As a result, the "listener" sent the signals irrespective of the activity that originated the request; i.e. we could not cause a module to suspend work on behalf of a particular activity while continuing work on behalf of another or to later resume work on behalf of a particular activity. Tagging of the code could have been simulated: The module's implementor/manager could (and in a real implementation should) maintain a list of activities in which the module was registered. Whenever the module was registered in a new activity its tag would be added to the list and whenever it was deregistered a tag would be removed from the list. The implementor could cycle through the list and run the module code accordingly. Of course, how and when to switch between activities is an open question. Actually providing this capability is clearly non-trivial, because separate context information would have to be maintained for each activity, (in all likelyhood, one would want a per-activity stack), an activity filterable communication channel and activity context switching provided for. In either case, it is beyond the scope of this thesis.

7.3.5 Signal Handlers

There is a noted deficiency in signal handling facilities within System V; namely, signal handlers could not execute system calls reliably. This posed somewhat of a problem in the handling of the suspend and resume handlers. It was my original intention that the handlers for these signals be reset immediately upon being caught. This is so that receipt of a second signal during the first, e.g. for objects registered in 2 or more activities, would not present a problem. Under the current implementation signal handlers were limited to setting a flag variable and the signals are reset to be caught based on this flag immediately thereafter in the context of the main program. Nevertheless, the possibility exists of signal problems if a second signal arrives prior to the re-setting of the handler within the main program. On the other hand, this was typically not the case in the simulation.

7.3.6 Object Implementation

Certain aspects of the object implementation in the simulation are inadequate in terms of a "real" system. This arose out of the attempt to reduce the complexity of simulation components other than the AC, and in particular to reduce the complexity of objects by eliminating "watchers". This is particularly relevant in the case of object and ACM deaths. When a module dies a problem arises in distinguishing between ACM's and other modules (or objects). In a "real" system the Object Implementor (perhaps via the module's "watcher") would have to make the distinction. For purposes of the simulation, object and acm records were kept in separate database files, with each maintaining sufficient information and key fields to be able to "kill" the module (which means knowing its process id), as well as to notify the interested parties. For ACM's in particular, we needed to be able to determine the activity that it controlled. Whereas objects have owning activity tag information, ACM's do not. Therefore, the "owning" activity name was stored. This represented somewhat of a departure from the original philosophy of keeping name information out of the object implementor database and relying solely on id's and implementor/manager port to index the records. Had the goal of this thesis been to implement a more sophisticated object implementor, I would have introduced "watcher" processes [Mayott88]. A watcher would be a process that is forked along with the object and whose sole purpose is to wait for the death of the object. These watchers would have knowledge of the "type" of module they oversaw. They would notify the Object Implementor of the death of objects. In the current implementation, objects themselves handle the notification.

In order to maintain ACM information by tag (as for other objects/modules) additional communication would have been required between the AC and the Object Implementor. At the time that an activity was created, the Object Implementor would be notified of the activity tag that was associated with the ACM. The ACM file in the database would then be updated to contain the tag instead of the name of the activity. This would have complicated the implementation of both the AC and the Object Implementor, especially the database implementation of the latter.

Chapter Eight

Conclusions, Related Topics, and Future Work

8.1 Conclusions

The goal of this thesis was to provide a functional, distributed Activity Coordinator that was able to handle at least the minimal set of operations put forth originally in Heliotis' thesis and described here in Chapter 5. This has been accomplished, with limitations as noted in the preceding chapter. Suggestions for further enhancements that could be provided for a "real" Activity Coordinator are provided below. In addition, other activity system components were developed to test the AC and to gain some insight into programming in an activity-based environment. To a lesser degree, these components represent a basis for a "real" activity system. Despite the relatively primitive implementation of these elements, the simulation, in conjunction with use of the test tools, demonstrates the usefulness of activities in developing and managing distributed systems.

8.1.1 Activity-based Programming

Certain programming practices arose in the development of the simulation that I feel are significant, and show the usefulness of the activity system as an aid to writing distributed programs.

1. The hierarchical nature of activities/sub-activities/objects encouraged top down design and implementation at the system level. In virtually all instances, it represented the only "reasonable" way to create the system, because typically a parent ACM knew either internally or at least locally all of the information required to initiate its child activities, and likewise objects that it owned. This can readily be seen in the startup of the simulation (section 6.6). This, in turn, provides structure and order to a number of concurrent events that otherwise may not have proceeded in an orderly manner. Again, I would refer the reader to the section on the simulation startup, and note especially the description of the event sequence.

2. Development of individual object implementors/managers and ACM's was greatly simplified by the existence of the activity system, and particularly the underlying communications support. Rather than creating com-

munications handlers within each module, one simply specified the actions to be taken upon receipt of a particular message, and set up a status message in reply. For purposes of the simulation at least, one could create a new module by making trivial modifications to a prototype version. The fact that the creation and startup of the module was in part off-loaded to the Object Implementor also simplified the programming of objects and ACM's.

8.1.2 Activity-based Distributed Systems Management

The system developed to test the AC clearly demonstrates how the use of activities contributes to the effective management of a distributed system. As noted above, activity commands help to bring the system up in an orderly manner. Once the system is started, the set of objects participating in a given activity as well as the state of these objects is easily controlled via a small working set of activity commands. If a module is killed, all necessary "interested parties" are notified, and the appropriate actions can be taken. Notification can be used not only with respect to failures, but also in terms of synchronization; as, for example, when the work of a group of objects is suspended.

8.2 Issues and Related Topics

8.2.1 Atomic Actions

One of the questions raised with respect to activities in the original thesis ([Heliotis84]) was "whether atomicity is just another application of activities, or if it is actually a more basic ... protocol in the lower levels of the operating system" [Heliotis84]. Having implemented a functional, but not fault-tolerant Activity Coordinator, I would say that atomic transactions are indeed required at a lower level. There are several reasons for this.

First, within the Activity Coordinator database itself, there are no guarantees of atomicity of updates. Consider for example the creation of an object. This operation requires updates to two database files, the object file (where a new record must be added) and the activities file (where the object must be added to the registration list of its owner). Either of these operations may fail independently of the other. Under the current implementation, the operation will fail, but it is possible that the updates to the database will remain. The "ophandler" provides a certain amount of "undo-redo" code for cases like this, but there is no guarantee that this will be effective: it must be

implemented at a lower level. In order to provide true consistency of the database this operation should be viewed as an atomic transaction at the level of the database manager; i.e. if either update fails not only should the operation fail, but the entire transaction should be undone, so that the database is returned to a consistent state. A step in the right direction, though by no means the complete solution would be to implement an atomic transaction scheme within the database manager used by the AC.

Second, although the "user" that requests an activity operation ultimately sees whether a command succeeded or failed, the objects/ACM's that are involved in the command do not. Consider the sending of an activity command. An activity command will fail if any member of the tree rooted at the destination fails to respond to the message. Under the current implementation, the command will not be sent to any ancestors of the failed recipient. However, any peers, peer's descendants and/or descendants of this recipient may well have already executed the command. One solution is to designate the AC as the coordinator for the transaction, and to execute the activity command using a 2-phase commit protocol determined by the "user" amongst the ACM's and object implementors/managers. In this case, additional activity commands for 2-phase commit primitives Sync (Precommit), Commit, and Abort would be added to the AC. In addition, all ACM's and object implementors/managers would have to have code for handling this 2-phase commit protocol. Of course, this applies not only to the sending of activity commands but to all other AC operations as well. A better solution would be to make these primitives available to the AC via the operating system. Again, the AC could be designated as the coordinator for the transaction, but instead of the user being responsible for providing the protocol, the AC would control the 2-phase commit for activity operations. Granted, this takes away some of the flexibility of activity commands, but it also relieves the user of the burden of assuring that the commands are carried out correctly.

Finally, there are a number of instances where the Activity Coordinator, Object Implementor, and Name Server databases must all be updated. No guarantees are made concerning the consistency of these databases when viewed as a whole. For example, it may well be that an object is created and registered with both the Activity Coordinator and Object Implementor, but not with the Name Server. These updates should also have the capability of being performed as atomic transactions, although this is clearly less important than guaranteeing the consistency of the individual databases.

8.2.2 Real-Time Systems

The question remains as to the appropriateness of Activity System for real-time applications. The current implementation is clearly deficient in a number of areas with respect to real-time requirements, owing to the simplified nature of this initial implementation. However, even assuming the enhancements described in the following sections were implemented, it is not clear that the current test-bed is adequate for making determinations with respect to activities and real-time systems. UNIX(tm) is, after all, not a real-time operating system, and accordingly one might want to consider porting activities to a different operating system. On the other hand, moving some of the functions of the activity system into the kernel could also be investigated. Ultimately, the true test of activities in a real-time environment can only come when we "move away from simulations on large computers, and start using small controller computers connected to real-world devices" [Heliotis84]. Also, there is the problem of obtaining timings for activity operations. In order to even begin to achieve any kind of time measurements, two issues previously mentioned need to be addressed: provision for real-time interval timers within System V (as are available in 4.2BSD) and providing timeout as a command argument for activity operations. Beyond this, system fluctuations in terms of other user processes that may be running and particularly in terms of network activity make obtaining accurate timings in the current setting virtually impossible. Even if these elements were eliminated, there are still dependencies on the state of the objects themselves, what operations are performed and the order in which they are executed. Clearly, a much more sophisticated set of test programs is needed.

8.3 System Performance

The above considerations notwithstanding, there are areas in which the performance of the implementation could be improved. These are primarily in terms of the AC database and the underlying communications required by all system components.

8.3.1 Communication Enhancements

Some of the inefficiency of the current implementation could be overcome by modifications to the underlying communications used by the AC. In order to communicate with a group of ACM's, Object Managers and Implemen-

tors, the AC presently uses a somewhat inefficient technique: Namely, an array of sockets is created and the send is done in a loop. A "select" is then performed to determine which receivers have replied. Thus, although broadcast capabilities are supported by the Ethernet as well as by tcp/ip, this implementation was forced to send messages to each processor individually. Performance could be greatly improved by introducing multicast or broadcast capabilities for use by the program that implements the AC. (see [Cook85], [Frank85]). At the transport level, the use of TCP/IP detracts somewhat from the performance of the system, due to the relatively large packet size employed [McQuillan77]. TCP/IP is a general purpose protocol that is intended for relatively large block data transfer as compared to the type of short message exchange required by the activity system. However, in order to guarantee reliable message delivery, TCP/IP was required. There are protocols that would presumably be more appropriate for the activity system is described in [Moore82]. This problem is less significant to the overall system performance than the lack of broadcast/multicast, but could be more easily addressed. Ultimately, one would probably want to develop a new protocol directly on top of raw sockets (transport endpoints) that was optimized for the type of communication required by the AC as well as other system components.

8.3.2 Database Enhancements

Another weakness in the current implementation involves the activity database. As noted in "Implementation Issues", the choice of C-Tree for the DBMS was made for the sake of expediancy. Like tcp/ip, it is a general purpose package that is not optimized for an activity based system. Alternatives to the current database implementation include:

(1) Variants of the B-tree index - Of particular interest are the Prefix B-Tree, because of its advantages in retrieval time, and B-Link Trees and PO-B Trees, because of their concurrency properties. [Fishbeck87]

(2) Extendible (Dynamic) hash files [Ellis83] - Dynamic hashing schemes seek to provide the advantages of hashing, namely fast lookup, while eliminating the primary problem, namely the degraded performance associated with hash table growth. Hash tables are virtually useless for sequential access, but since in the activity system sequential record retrieval is not required, the use of hash tables as opposed to B-trees appears to be a good alternative. Ellis has provided algorithms for concurrently accessible dynamic hash tables as well as a dis-

tributed version.

It would be interesting to see if the use of a particular B-Tree variant or Extendible hash files made a noticeable difference in the performance of the system.

(3) Use of shared memory - The primary weakness of the current database arises from the use of shared files rather than shared memory. The use of main-memory data structures (linked lists and trees) with a "shadow" image on disk would offer the best solution in terms of both real-time requirements and fault-tolerance. However, this would require significant changes to the kernel in terms of memory management and the file system. One such storage system is implemented as part of the Clouds virtual memory manager [Pitts86].

8.4 Future Work

Beyond simply improving the performance of the current implementation, there are areas in which we may examine extending the role of the Activity Coordinator. In addition, support for object development in the activity environment is an area that could be investigated.

8.4.1 Communications Monitoring

It has been suggested ([Mayott88]) that some type of communication monitoring facilities be provided as part of the AC. Information as to the network accessibility of a node could be kept by the AC so that, at the very least, the AC does not attempt commands to a machine that is known to be down. It is also possible that users as well as other servers could query the AC regarding the status of the network before attempting message exchanges. I would suggest a slightly different alternative; namely a communications monitoring activity. This Communications Monitor would be somewhat akin to the Communication Manager in TABS (see section 3.10) but with less involvement in the underlying communication mechanism itself and certainly no function with regard to object registration in activities. The Communications Monitor would provide, amongst other things, a default ACM for a "communications monitoring" activity. The Activity Coordinator, Name Server, and Object Implementor could then create various connections between objects as subactivities within the communications monitoring activity (and register the objects themselves in these activities). Each server would then furnish the Communications Monitor with notification of com-

munications failures. The Communications Monitor would be better equipped to analyze communication problems since it would be receiving information from 3 sources: The Object Implementor is responsible for Connection/Disconnection of objects and can provide information in this capacity that is not directly available to the Activity Coordinator. The Activity Coordinator must typically communicate with object implementors/ managers and ACM's, but also with other AC servers when the command pertains to a non-local object/activity. This would provide the Communications Monitor with status regarding AC servers as well as objects and ACM's. Each Name Server must communicate with other Name Servers for all database updates. Thus, it is potentially the best source of information regarding the network accessability of a node.

8.4.2 Fault-tolerance

We have already seen how the activity system provides for failure notification in the event of an object/ACM death, and how this in turn facilitates the orderly handling of the failure. At this point the recovery mechanism is left to the user. At first glance it would appear to be desirable to provide an automated recovery mechanism within the activity system itself. However, the very flexibility that is provided by the activity system makes this highly impractical. Consider first the death of an object. It would be easy enough for the Object Implementor to (1) maintain the pathname of the code modules to be executed for all objects and (2) re-create the object upon notification of its failure. But, this could very well pose a problem if any of the ACM's for activities in which the object was registered had likewise attempted to re-initiate the object. Moreover, the problem remains of how to re-register the object in its respective activities (since we have no way of knowing whether or not this is handled by the object implementor/manager itself). Similarly, there is the problem re-establishing connections with other objects. Even assuming that we could establish a "generic" scheme for transparently recovering an object to its startup state, there remains the question of how to bring the object forward to some "acceptable" state prior to its failure. Likewise, consider the failure of an ACM. The default action for the AC in the event of an ACM death could be modified so that it attempted to re-create the module and restart the activity. However, a problem similar to that for objects arises; namely, we do not know what the action of the parent ACM will be upon receiving notification of the death of its child. Again, provided we could bring the ACM back to its startup state, it would then have to be brought forward to an "acceptable" state prior to the failure. It is unclear how one might represent the state of an ACM so that this

could be accomplished effectively. The fact that an automated recovery scheme can not readily be provided should not be seen as an indictment of the activity system. Quite the contrary, it indicates the flexibility of the system in providing tools for failure recovery without mandating a particular mechanism.

One way to provide for higher reliability is through replication of resources. Underlying system support must then be provided to enable transparent access to these resources. Within the framework of an activity system, this could be facilitated through the addition of new activity commands, e.g. to Move an Object/Activity or to Replicate an Object/Activity. As noted in section 5.2, the AC would then have to be modified to provide for transparent access to such objects/activities. This is simplified in the current implementation by the fact that an object has a "home" node. Changes required within the Activity Coordinator are relatively straightforward compared to the much larger issue of how to represent an object/activity's internal state so that it may be effectively moved or replicated.

8.4.3 Object Implementation

Concerning the simulation of system components other than the AC (i.e. object managers/implementors and ACM's), tools for automated manager development (as in Cronus) would be highly advantageous to the implementation of a "real" activity system. While it was relatively easy to provide objects for the simulation, this was primarily because they were not expected to behave in a "realistic" fashion, but simply to react as required by the activity system (i.e., sufficient to test the AC). Any such tool would have to take into account the additional requirement of providing for activity context information as part of an object's implementor/manager (see section 7.3.4)

Also in the area of object development, investigation into the use of types/classes of object implementors is probably merited. Such projects could well be of interest independent of an activity system, but strictly in terms of object-oriented programming. On the other hand, it would be interesting to see what effect the use of Object Implementors for different object types would have in place of the single Object Implementor currently found in the Activity System.

Bibliography

[Ada83] Military Standard, Ada Programming Language, ANSI/MIL-STD-1815A, American National Standards Institute, January 1983.

[Alford85] M.W. Alford, et. al., "A Graph Model Based Approach to Specifications" Distributed Systems: Methods and Tools for Specification, Lecture Notes in Computer Science volume 190, Springer-Verlag, 1985.

[Allchin83] James Edward Allchin, "An Architecture For Reliable Decentralized Systems", PhD Thesis, Georgia Institute of Technology, 1983.

[ATT85] AT&T 3B2 Computer UNIX(tm) System V Release 2.0, Programmer Reference Manual, July 1985.

[Cook85] Robert P. Cook and Thomas J. LeBlanc, "High-Level Broadcast for Local Area Networks", *IEEE Software,* May 1985, p.40-48.

[Cox87] Brad Cox "Building malleable systems from software 'chips'" ComputerWorld, March 30, 1987., p.59-68.

[Dasgupta85] P. Dasgupta, R.J. LeBlanc Jr., E. Spafford, "The Clouds Project: Designing and Implementing a Fault Tolerant, Distributed Operating System", Georgia Institute of Technology, Technical Report: GIT-ICS-85/29, 1985.

[Ellis82] Carla S. Ellis, Jerome A. Feldman, and James E. Heliotis, "Language Constructs and Support Systems for Distributed Computing", University of Rochester Technical Report #102, May 1982.

[Ellis85] Carla S.Ellis and James E. Heliotis, "Structuring Resilient Distributed Programs with the Activity Model", University of Rochester Technical Report #163, June 1985.

[Faircom87] C-Tree Programmer's Guide, Fourth Edition Version 4.1 Release F, 1987.

[Fishbeck87] Sally E. Fishbeck, "The Ubiquitous B-Tree, Volume II", Master Thesis, Rochester Institute of Technology, 1987.

[Frank85] Ariel J. Frank, Larry D. Wittie, and Arthur J. Bernstein, "Multicast Communication on Network Computers", *IEEE Software,* May 1985, p.49-60.

[Gurwitz86] Robert Gurwitz, Michael A. Dean and Richard E. Schantz, "Programming Support in the Cronus Distributed Operating System", IEEE Computer Society, 1986.

[Heliotis84] James E. Heliotis, "Language Constructs for the Management of Distributed Computations", PhD Thesis, University of Rochester, 1984.

[Hommel85] Gunter Hommel, "Language Constructs for Distributed Programs", in *Distributed Systems: Methods and Tools for Specification,* Lecture Notes in Computer Science volume 190, Springer-Verlag, 1985.

[Jones79] Anita K. Jones, "The Object Model: A Conceptual Tool for Structuring Software" in *Operating Systems: An Advanced Course* Lecture Notes in Computer Science volume 190, Springer-Verlag, 1979, p.7-16.

[Joy83] Joy, William N., et. al. "4.2BSD System Manual", University of California Berkeley, Report No. UCB/CSD 83/???, July 1983.

[Kramer85] Jeff Kramer and Jeff Magee, "Dynamic Configuration For Distributed Systems" *IEEE Transactions On Software Engineering,* Vol.11, No.4, p.424-436, April 1985.

[Kramer84] Jeff Kramer, Jeff Magee, and Morris Sloman "A Software Architecture for Distributed Computer Control Systems" *Automatica* Vol.20, No.1, p.93-102, April 1984.

[Kramer83] Jeff Kramer, et. al. "CONIC: An Integrated Approach to Distributed Computer Control Systems" *IEE Proceedings* Vol.130, Part E, No.1, p.1-10, January 1983.

[Lagally79] K. Lagally, "Synchronization in a Layered System", in *Operating Systems: An Advanced Course* Lecture Notes in Computer Science volume 190, Springer-Verlag, 1979, p.252-278

[Lamport 78] Leslie Lamport, "Time, Clocks and the Ordering of Events in a Distributed System", *Communications of the ACM,* July 1978, p.558-565.

[Lamport 82] Leslie Lamport and P.M. Melliar-Smith "Synchronizing Clocks in the Presence of Faults", SRI International CSL, March 1982.

[LeBlanc85] Thomas A. LeBlanc and Stuart J. Friedburg, "Hierarchical Process Composition in Distributed Operating Systems", *IEEE Distributed Computing Systems 1985,* p.26-34.

[Leffler83a] Leffler, Samuel J., Fabry, Robert S. and Joy, William N., "A 4.2BSD Interprocess Communication Primer", University of California Berkeley, Report No. UCB/CSD 83/145, July 1983.

[Leffler83b] Leffler, Samuel J., Fabry, Robert S. and Joy, William N., "4.2BSD Networking Implementation Notes" University of California Berkeley, Report No. UCB/CSD 83/???, July 1983.

[Lewis87] Scott Lewis "B-tree filing systems for C" *Computer Language* , Vol. 4, Number 8, August 1987, p.113-122.

[Liskov79] Barbara Liskov, "Primitives for Distributed Computing", *Communications of the ACM* ,May 1979, p.33-42.

[Liskov82] Barbara Liskov, "On Linguistic Support for Distributed Programs", *IEEE Transactions on Software Engineering,* Vol. SE-8, No. 3, May 1982, p.203-210.

[Liskov85] Barbara Liskov, "The Argus Language and System", in Distributed Systems: Methods and Tools for Specification, Lecture Notes in Computer Science volume 190, Springer-Verlag, 1985.

[Marzullo85] Keith Marzullo "Maintaining the Time in a Distributed System" *Operating Systems Review,* Vol.19, No.3, July 1985, p.44-55.

[Mayott88] Stewart Mayott, "Implementation of a Module Implementor for an Activity-Based Distributed System" Master Thesis, Rochester Institute of Technology, 1988.

[McManis87] Chuck McManis, "System V and SunOS" Sun Microsystems, 1987.

[McQuillan77] John M. McQuillan and David C. Walden "The ARPA Network Design Decisions" *Computer Networks,* North-Holland Publishing Company, 1977. p.243-289.

[Moore82] Lee C. Moore, Liudvikas Bukys, and James E. Heliotis, "Design and Implementation of a Local Network Message Passing Protocol", *IEEE ???,* 1982, p.70-74.

[Moss85] J. Eliot B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing,* MIT Press, 1985.

[Mueller83] Mueller, Erik T., Moore, Johanna D., and Popek, Gerald J., "A Nested Transaction Mechanism for LOCUS", *Proceedings of the Ninth Symposium on Operating System Principles, ACM/SIGOPS, Operating Systems*

Review, Vol.17 No. 5, October, 1983, p.71-89.

[Natarajan85] N. Natarajan "Communication And Synchronization Primitives For Distributed Programs", *IEEE Transactions on Software Engineering,* Vol.11, No.4, p.396-416, 1985.

[Papadimitriou86] Christos Papadimitriou, *Database Concurrency Control,* Computer Science Press, 1986.

[Pitts86] David Vernon Pitts, "A Storage Management System For A Reliable Distributed Operating System", PhD Thesis, Georgia Institute of Technology, 1986.

[Popek81] Gerald Popek, et. al., "LOCUS: A Network Transparent, High Reliability Distributed System", *Proceedings of the Eighth Symposium on Operating System Principles, ACM/SIGOPS,* Operating Systems Review, Vol.15 No. 5, December, 1981, p.169-177.

[Rashid80] Richard F. Rashid, "An Inter-Process Communication Facility for UNIX", Carnegie-Mellon University Technical Report CMU-CS-80-124, June 1980.

[Rashid81] Richard F. Rashid and George G. Robinson, "Accent: A communication oriented network operating system kernel", *Communications of the ACM,* Vol. ??, No.?, p.64-75, 1981.

[Schantz86] Richard E. Schantz, Robert H. Thomas and Girome Bono, "The Architecture of the Cronus Distributed Operating System", IEEE Computer Society, 1986.

[Schwan86] K. Schwan, A.K. Jones, "Flexible Software Development For Multiple Computer-Systems", *IEEE Transactions On Software Engineering,* Vol.12, No.3, p. 385-401, 1986.

[Spafford86] Eugene Howard Spafford "Kernel Structures For A Distributed Operating System", PhD Thesis, Georgia Institute of Technology, 1986.

[Spector??] Alfred Z. Spector and Peter M. Schwarz, "Transactions: A Construct for Reliable Distributed Computing", p.18-35.

[Spector85] A.Z. Spector, J. Butcher, D.S. Daniels, et al. "Support For Distributed Transactions In The TABS Prototype", *IEEE Transactions On Software Engineering,* Vol 11, Num 6, p.520-530, 1985.

[Sha83] Lui Sha, et. al., "Distributed Co-operating Processes and Transactions", in *Distributed Computing Systems: Synchronization, Control and Communication,* Y. Paker and J.P. Verjus, editors, Academic Press, 1983.

[Taylor86] D.J.Taylor, "Concurrency And Forward Recovery In Atomic Actions", IEEE Transactions On Software Engineering, V12, N1, P69-78, 1986

[Walker83] Bruce Walker, et. al., "The LOCUS Distributed Operating System", *Proceedings of the Ninth Symposium on Operating System Principles, ACM/SIGOPS, Operating Systems Review,* Vol.17 No. 5, October, 1983, p.49-70.

[Wilkes85] Wilkes, C. Thomas, and LeBlanc, Richard J., "Systems Programming with Objects and Actions", *IEEE Distributed Computing Systems, 1985.,* p.132-139.

[Wirth76] Nicklaus Wirth *Algorithms + Data Structures = Programs* , Prentice-Hall, 1976.

[Xerox81] Courier: The Remote Procedure Call Protocol *Xerox System Integration Standard,* Xerox Corporation, December 1981.

Appendix A

Activity Coordinator
Function Descriptions

File: switch.c

Description:
This file contains the main server portion of the Actvivity Coordinator.

Functions:

ac_switch()

The ac_switch listens at a well known address (as defined in ac.h). It accepts connections from user modules, object implementors/managers, and ACM's and forks a msg_handler process to handle the request. The socket returned by accepting the connection is passed on to the msg_handler as socket_svc.

msg_handler(sockt_svc)
    int sockt_svc;

The msg_handler reads requests on sockt_svc (passed to it by ac_switchboard) It parses the message and invokes an op_handler to perform the requested operation (see op_handler.doc). After the op_handler completes the request the msg_handler forwards the result to the requesting module on sockt_svc

returns:
    0 if the requested operation succeeded
    -1 if the requested operation failed

Note that at this point the AC does nothing with this information. Future, more "intelligent", versions of the AC could take a more active part in handling failed operations.

File: op_handler.c

These are the handlers for each of the operations defined in the activity system. For each function there is a reply returned. This reply is forwarded to the requesting module by the msg_handler. Note that instead of using command arguments (parameters), each routine does a sscanf on "request" to get the arguments that it needs. This was done so that if these functions were to be used as modules in a message passing environment, minimal modification would be required.

Functions:

create_obj - create a new object and register it in its owning activity
register_obj - register and object in an activity other than its owner
deregister_obj - deregister an object from an activity
invalidate_obj - invalidate an id; used by remove_obj and destroy_obj
remove_obj - "peaceful" object destruction ("normal" termination)
destroy_obj - "violent" object destruction ("kill")
get_imp_port - get an object's implementor port
get_mgr_port - get an object's manager port
get_owner_tag - get an object's owning activity tag
get_obj_role - get an object's role
get_acm_port - get an activity's ACM port
get_act_status - get an activity's status
get_act_parent - get an activity's parent tag
create_act - create a new activity
act_cmd_send - send an activity command
acm_handles - block a subactivity, object or group of objects from receiving an activity command
acm_failed - an ACM failed or was terminated
send_obj_cmd - send a command directly to an object
ac_dereg_obj - deregister an object from an activity without notifying the implementor and manager of the object

```
create_obj(request, reply)
        char *request, *reply;

        request -> owner_tag, role, impl_port, mgr_port
        char owner_tag[TAGLEN], impl_port[PORTLEN], mgr_port[PORTLEN];
        int role;

        reply <- op, status, new_id
        int op, status;
        char new_id[IDLEN],
```

Provided the owner_tag is for an activity that currently exists in the database, a new id is created for the object (see get_id()). A REGISTER_OBJ message is sent to the implementor and manager for the object to be created, at the ports supplied in the request. If the object implementor returns a "STATUS_OK" reply within the required timeout and the id matches that of the request, the object record is added to the activity coordinator's database.

returns: 0 = successful operation
-1 = failed

```
register_obj(request, reply)
        char *request, *reply;

        request -> id, tag
        char id[IDLEN], tag[TAGLEN];

        reply <- op, status, id, tag
        int op, status;
        char id[IDLEN], tag[TAGLEN];
```

If the id is for an object that exists in the database and
the tag is for an existing activity, send a "REGISTER_OBJ" msg together with
the id and tag to the mgr_port & imp_port of the object.
Wait for a reply from both.
Provided both return a "STATUS_OK" reply, the id is added to the object list
of the activity and the tag is added to the activity list of the object.

```
deregister_obj(request, reply)
        char *request, *reply;

        request -> id, tag
        char id[IDLEN], tag[TAGLEN];

        reply <- op, status, id, tag
        int op, status;
        char id[IDLEN], tag[TAGLEN];
```

If both the object and the activity are found,
send a "DEREGISTER_OBJ" message along with the id and tag to the
implementor and manager ports of the object, and wait for a reply from both.
Provided a "STATUS_OK" reply is returned within TIMEOUT seconds,
the id is removed from the object list of the activity,
and the tag is removed from the activity registration list of the object.

```
invalidate_obj(id, objptr)
        char id[IDLEN];
        OBJREC *objptr;
```

This routine is called by both remove_obj and destroy_obj; it is not intended to be a function that the user would invoke.

If the object is found in the database, the message "DEREGISTER_OBJ" is sent to the object's implementor and manager ports,
along with the id and a special tag indicating that the object is to be deregistered from all activities. If the command is invoked by remove_obj then the routine waits for a reply; if it is invoked by destroy_obj, no reply is expected.

Returns: 0 = successful operation
        -1 = failed

```
remove_obj(request, reply)
        char *request, *reply;

        request -> id
        char id[IDLEN]

        reply <- op, status, id
        int op, status;
        char id[IDLEN];
```

If the object id is invalidated (see invalidate_obj),
remove the object from each of its activities; i.e.
for each activity in the object's actvity list,
remove the object id from the activity's object list
(This is rather a time consuming operation and it is quite possible
that several retries will be needed to update all the necessary records
in the database)
When the object has been deregistered from each of its activities,
delete the entry from the database.

```
destroy_obj(request, reply)
        char *request, *reply;

        request -> id
        char id[IDLEN]

        reply <- op, status, id
        int op, status;
        char id[IDLEN];
```

If the object id is invalidated (see invalidate_obj),
remove the object from each of its activities; i.e.
for each activity in the object's actvity list,
remove the object id from the activity's object list
Send "OBJ_DIED" notices along with the id to each activity's ACM
do not wait for a reply.
When the object has been deregistered from each of its activities,
delete the entry from the database.

Returns: 0 = successful operation
        -1 = failed

```
get_imp_port(request, reply)
        char *request, *reply;

        request -> id
        char id[IDLEN];

        reply <- op, id, status, imp_port
        int op, status;
        char id[IDLEN], imp_port[PORTLEN];
```

If the object indicated by "id" is found, set status = STATUS_OK, and return the implementor port in reply.

```
get_mgr_port(request, reply)
        char *request, *reply;

        request -> id
        char id[IDLEN]

        reply <- op, id, status, mgr_port
        int op, status;
        char id[IDLEN], mgr_port[PORTLEN];
```

If the object indicated by "id" is found, set status = STATUS_OK, and return the manager port in reply

```
get_owner_tag(request, reply)
        char *request, *reply;

        request -> id
        char id[IDLEN];

        reply <- op, id, status, owner_tag
        int op, status;
        char id[IDLEN], owner_tag[TAGLEN];
```

If the object indicated by "id" is found, set status = STATUS_OK, and return the tag of the owning activity in reply

```
get_obj_role(request, reply)
        char *request, *reply;

        request -> id
        char id[IDLEN];

        reply <- op, id, status, role
        int op, status, role;
        char id[IDLEN];
```

If the object indicated by "id" is found, set status = STATUS_OK, and return the role in reply

```
get_acm_port(request, reply)
        char *request, *reply;

        request -> tag
        char tag[TAGLEN];

        reply <- op, tag, status, acm_port
        char acm_port[PORTLEN];
```

If the activity indicated by "tag" is found, set status = STATUS_OK, and return the port of the Activity Control Module in reply

```
get_act_staus(request, reply)
        char *request, *reply;

        request -> tag
        char tag[TAGLEN];

        reply <- op, tag, status, act_status
        int acm_status;
```

If the activity indicated by "tag" is found, set status = STATUS_OK, and return the status of the activity in reply

```
get_act_parent(request, reply)
        char *request, *reply;

        request -> tag
        char tag[TAGLEN];

        reply <- op, tag, status, act_parent
        char act_parent[TAGLEN];
```

If the activity indicated by "tag" is found, set status = STATUS_OK, and return the tag of the parent activity in reply

```
create_act(request, reply)
        char *request, *reply;

        request -> parent_tag, acm_port
        char parent_tag[TAGLEN], acm_port[PORTLEN];

        reply <- op, status, newtag
        int op, status;
        char newtag[TAGLEN];
```

Create a new activity with parent specified by parent_tag and with an ACM whose port is acm_port. If parent_tag is null create a new root activity; otherwise, add the new activity as a child of the parent (see add_child_act) If the acm_port is null give the activity the default ACM whose port is specified in ACM_DEFAULT (see ac.h).

```
act_cmd_send(request, reply)
        char *request, *reply;

        request -> tag, msg
        char tag[TAGLEN];
        int msg;

        reply <- op, tag, status
        int op, status;
        char tag[TAGLEN];
```

Send an Activity Command; causes the command whose number is indicated by "msg"
to be propagated to all sub-activities and objects in the tree rooted
at a specified activity, as well as to the activity itself.
The message is sent to each of the sub-activities recursively, thus effecting
a bottom-up order. The message is then sent to the implementor and manager
of all objects registered in the activity.
Objects and/or activities that are marked as being "handled" (see acm_handles())
by a parent activity do not receive the message.
The following "system defined" activity commands are recognized
by the Activity Coordinator:

Terminate - Upon receiving notice that the ACM has processed the command,
the coordinator invalidates the activity tag and sends an "ACM TERMINATED"
notice to the parent activity.
If the command is directed to a root activity,
the "Terminate" command that is normally sent to object implementors/managers
is replaced by a "Deregister" command.

Suspend - The "status" of the activity is set to ACT_SUSPENDED

Free - The "status" of the activity is set to ACT_WORKING

```
acm_handles(request, reply)
        char *request, *reply;

        request -> op, msg, id, tag, role
        reply <- op, status, msg
```

ACM "Handles" Messages - blocks one or more participants in an activity from receiving activity commands. This command may be applied to a sub-activity, object, or set of objects with the same role.
The coordinator marks the corresponding sub-activity/objects as not receiving a particular command. This is done using a bit mask (associated with each sub-activity/object) to select which commands are to be blocked. The bit mask is determined by "or'ing" the message values found in msg.h

```
acm_failed(request, reply)
        char *request, *reply;

        request -> op, tag
        char tag[TAGLEN];

        reply <- op, status
```

If the tag is for an activity that exists in the database,
the AC sends an ACM Failed notice to the parent activity
and ACT_TERMINATE to all acm's for the activity tree rooted at "tag",
using act_cmd_send(tag, ACT_TERMINATE).

```
send_obj_cmd(request, reply)
        char *request, *reply

request -> op, cmd, msgbuf, id, replyflag, timeout
reply <- op, cmd, status, id

send a command "cmd msgbuf" to the object "id".
If replyflag = 1 wait for replies for "timeout" seconds.
If replyflag = 0 don't wait for replies (in which case timeout is ignored).
```

```
ac_dereg_obj(request, reply)
        char *request, *reply
```

request -> op, id, tag
reply <- op, status, id

deregister object "id" from activity "tag", without sending DEREGISTER commands to the implementor/mgr.

File: db2.c

Description:

Intermediate database functions used by the op_handlers. These functions call c-tree routines in order to do the actual update of the database. Typically, these functions set up the buffers that are needed by c-tree to do the required updates. Also, c-tree routines may fail for a variety of reasons. If it is possible to handle the failure, it will be done in these routines.

The most common source of failure of a c-tree routine is that we are unable to get a lock on a data record we need. This simply means we need to back off and try again. Another common source of failure is that another process has updated a record between the time we made the initial read and the time we are ready to write the update back to disk. In this case we need to re-read the record and try again. Various methods for performing "multi-user" updates and handling deadlock can be implemented in these functions. The basic algorithm used in the intermediate database functions for performing updates is described in section 7.1.5.

The main motivation for having intermediate functions was to permit use of another DBMS (besides c-tree) without requiring any changes to the op_handler routines. Also, these routines should themselves require minimal modification in the event the underlying DBMS were changed; i.e., the parameters passed by these functions to the c-tree routines should be (more than) sufficient to achieve the desired results The only change that should be required would be to replace the include statements and in some cases actually reduce the number of buffers that are passed to the DBMS as parameters.

Functions:
add_obj_db - add an object record to the database
add_act_db - add an activity record to the database
add_actlist - add a tag to the activity registration list of an object.
add_objlist - add an id to the object registration list of an activity
get_obj - retrieve an object record from the database
get_act - retrieve an activity record from the database
rm_obj_db - remove an object record from the database
rm_act_db - remove an activity record from the database
rm_objlist - remove an id from the object registration list of an activity
rm_actlist - remove a tag from the activity registration list of an object
update_obj - update an object record.
update_act - update an activity record.
add_child_act - add a new child to the activity tree of the parent activity, and add the new child record to the database
rm_child_act - remove a child actitvity from the activity tree of its parent and delete the child activity record from the datab
copy_act - copy the current activity record into a new activity buffer
copy_obj - copy the current object record into a new object buffer
rmt_add_objlist - add an id to the object registration list of an activity as requested by a remote AC server.
rmt_add_actlist - add a tag to the activity registration list of an object as requested by a remote AC server.
rmt_rm_objlist - remove an id from the object registration list of an activity as requested by a remote AC server.
rmt_rm_actlist - remove a tag from the activity registration list of an object as requested by a remote AC server.

```
add_obj_db(objptr)
        char *objptr;
```

Add an object record to the database

returns: 0 = successful operation
        -1 = failed

```
add_act_db(actptr)
        char *actptr;
```

Add an activity record to the database

returns: 0 = successful operation
        -1 = failed

```
get_obj(id, objptr)
        char *id, *objptr;
```

Retrieve an object record with key "id" from the database into objptr

returns: 0 = successful operation
        -1 = failed

```
get_act(tag, actptr)
        char *tag, *actptr;
```

Retrieve an activity record with key "tag" from the database into actptr

returns: 0 = successful operation
        -1 = failed

```
rm_obj_db(id, cur_obj, chk_obj)
        char *id, *cur_obj, *chk_obj;
```

Remove object record with key "id" from the database.
The record is first read into the "cur_obj" buffer prior to being deleted.
The "chk_obj" buffer is needed to ensure that another process is not currently updating the record that we are trying to delete.

returns: 0 = successful operation
-1 = failed

```
rm_act_db(cur_act, chk_act)
        ACTREC *cur_act, *chk_act;
```

Remove activity record with key "tag" from the database.
The record is first read into the "cur_act" buffer prior to being deleted.
The "chk_act" buffer is required to ensure that another process is not currently updating the record that we are trying to delete.

returns: 0 = successful operation
-1 = failed

update_obj(cur_obj, new_obj, chk_obj)
char *cur_obj, *new_obj, *chk_obj;

Update an object record. The buffer "cur_obj" points to the "current ISAM record". The buffer "new_obj" points to the updated version of the record. The buffer "chk_obj" is used to detect concurrent update interference.

returns: 0 = successful operation
-1 = failed

update_act(cur_act, new_act, chk_act)
char *cur_act, *new_act, *chk_act;

Update an activity record. cur_act points to the "current ISAM record" buffer. new_act points to the updated version of the record. chk_act is used to detect multi-user update interference.

returns: 0 = successful operation
-1 = failed

```
add_objlist(id, imp_port, mgr_port, cur_act, new_act)
        char *id, *imp_port, *mgr_port;
        ACTREC *new_act, *cur_act;
```

Add id and port information to the object registration list of an activity cur_act points to the "current ISAM record" buffer. new_act points to the updated version of the record. This function follows the algorithm described in section 7.1.5 for performing updates.

returns: 0 = successful operation
-1 = failed

```
add_actlist(tag, cur_obj, new_obj)
        char *tag;
        OBJREC *new_obj, *cur_obj;
```

Add a tag to the activity registration list of an object. cur_obj points to the "current ISAM record" buffer. new_obj points to the updated version of the record. This function follows the algorithm described in section 7.1.5 for performing updates.

returns: 0 = successful operation
-1 = failed

rm_objlist(id, cur_act, new_act)
char *id;
ACTREC *new_act, *cur_act;

Remove an id from the object registration list of an activity cur_act points to the "current ISAM record" buffer. new_act points to the updated version of the record.

returns: 0 = successful operation
-1 = failed

rm_actlist(tag, cur_obj, new_obj)
char *tag;
OBJREC *new_obj, *cur_obj;

Remove a tag from the activity registration list of an object cur_obj points to the "current ISAM record" buffer. new_obj points to the updated version of the record.

returns: 0 = successful operation
-1 = failed

add_child_act(new_add, parent_tag, cur_parent, new_parent, cur_child, new_child)

```
char *parent_tag;
ACTREC *new_add, *cur_child, *new_child, *new_parent, *cur_parent;
```

Add a new child "new_add" to the activity tree of the parent activity, and add the new_add record to the database. If new_add is not the first child of the parent activity update the current child.

rm_child_act(cur_rec,cur_lsib,new_lsib,cur_rsib,new_rsib,cur_parent,new_parent)

```
ACTREC *cur_rec,*cur_lsib,*new_lsib,
        *cur_rsib,*new_rsib,*new_parent,*cur_parent;
```

Remove child_act from the activity tree of parent_act.

```
copy_act(cur_act, new_act, except)
        ACTREC *cur_act, *new_act;
        int except;
```

Make a copy of cur_act into new_act with the exeption of fields flagged in except.
Values of except are:

NO_EXCEPT 0
XOBJLIST 2
XACTPARENT 4
XACTCHILD 8
XACTLSIB 16
XACTRSIB 32

```
copy_obj(cur_obj, new_obj, except)
        OBJREC *cur_obj, *new_obj;
        int except;
```

Make a copy of cur_obj into new_obj with the exeption of fields flagged in except.
Values of except are:

NO_EXCEPT 0
XACTLIST 1

Note: the following are operations that are requested by a remote AC server when the object/activity in question can not be found locally.

```
rmt_add_objlist(request, reply)
        char *request, *reply;
        request -> op, id, obj_imp_port, obj_mgr_port, tag
        reply <-  op, status, id, tag
```

Add the id and port information to the object registration list of an activity as requested by a remote AC. After calling get_act with the tag provided, add_objlist is called as for a local activity, with the id and implementor/manager ports provided.

```
rmt_add_actlist(request, reply)
        char *request, *reply;

        request -> op, id, tag, act_acm
        reply <- op, status, id, tag
```

Add tag and acm port information to the activity registration list of an object as requested by a remote AC. After calling get_obj with the id provided, add_actlist is called as for a local activity, with the tag and acm port provided.

```
rmt_rm_objlist(request, reply)
        char *request, *reply;
        request -> op, id, tag
        reply <- op, status, id, tag
```

Remove an object from the registration list of an activity as requested by a remote AC. After calling get_act with the tag provided, rm_objlist is called as for a local activity, with the id provided.

```
rmt_rm_actlist(request, reply)
        char *request, *reply;
        request -> op, id, tag
        reply <- op, status, id, tag
```

Remove a activity from the registration list of an object as requested by a remote AC. After calling get_obj with the id provided, rm_actlist is called as for a local object, with the tag provided.

File: db_ct.c

Description:

Routines for manipulating the activity and object databases using c-tree. These functions are called by the intermediate database routines (see db2.c)

Functions:

get_rec(ifilenum, keyval, recptr)
    COUNT ifilenum;
    TEXT *keyval, *recptr;

Get a record with key "keyval" from the data file whose index file is ifilenum and store in recptr.

returns:

NO_ERROR: Successful operation
DNUL_ERR: recptr is null
DLOK_ERR: Could not get data record lock
INOT_ERR: Key value not found

COUNT update_rec(datnum, cur_buff, upd_buff, chk_buff)
    COUNT datnum;
    TEXT *cur_buff, *upd_buff, *chk_buff;

Update a record in data file "datnum". "cur_buff" contains the "current ISAM record" (see [Faircom87]). "upd_buff" contains the updated record. "chk_buff" is used to check for multi-user(process) update interference

returns:

NO_ERROR: Successful operation
KDUP_ERR: duplicate key value
IPND_ERR: record locks still pending
INOL_ERR: no room in c-tree's internal lock table
    (increase the MAX_LOCKS parameter in ctoptn.h)
DNUL_ERR: recptr is null
DLOK_ERR: Could not get data record lock
ICUR_ERR: No current ISAM record
ITIM_ERR: Record deleted by another process
XSIM_ERR: Simultaneous update interference

```
COUNT delete_rec(datnum, cur_buff, chk_buff)
        COUNT datnum;
        TEXT *cur_buff, *chk_buff;

description:
delete record "cur_buff" from data file "datnum"
chk_buff is used to check for multi-user(process) update interference
(For some reason c-tree requires this buffer to be global)

returns:
                NO_ERROR: Successful operation
                IPND_ERR: record locks still pending
                INOL_ERR: no room in c-tree's internal lock table
                        (increase the MAX_LOCKS parameter in ctoptn.h)
                DNUL_ERR: recptr is null
                DLOK_ERR: Could not get data record lock
                ICUR_ERR: No current ISAM record
                ITIM_ERR: Record already deleted by another process
                KDEL_ERR: could not delete key value
                XSIM_ERR: Simultaneous update interference

add_rec(datnum, recptr)
        COUNT datnum;
        TEXT *recptr;

add a new fixed length data record to data file "datnum" and add
key value to corresponding index file

returns:
        NO_ERROR: Successful operation
        KDUP_ERR: Duplicate key value
        DELFLG_ERR: Previous data record not deleted
        DNUL_ERR: Recptr is null
        WRITE_ERR: Directory is full
        FMOD_ERR: Data file mode error
        IPND_ERR: Other record locks pending
        INOL_ERR: Internal lock table full


create_db(dbdname)
        TEXT *dbdname;

Create a database in c-tree using the "ISAM" functions and
a parameter file "dbdname" to build all the files
(See [Fair87] for a description of parameter files)

returns:
        NO_ERROR: Successful operation
        XCRAT_ERR: Files could not be created
        XCLS_ERR: Files could not be closed
        XOPN: Files could not be opened
```

File: ipc2.c

Description:
Intermediate routines used by the op_handlers to set up the necessary
message data structures for the ipc routines.

At one point, message status information was contained in the activity
and object records. Since we do not want to have to lock out
other op_handler routines in order to send msgs concerning a
particular activity, these routines have since been modified to account
for the removal of the msg_sent and msg_recv fields from these records.
They have been replaced with MSG_LIST_NODES (see below)
that contain the appropriate port information in addition to message status.

Also we must now account for the fact that we
need to retrieve a record into act_rec based on tag, rather than having
a pointer to it in memory. Rather than re-constructing the tree in memory
it is traversed via disk reads and an array of linked lists constructed.
While this does not produce exactly the same order as that of an in-memory
traversal via pointers, it does achieve the desired result; namely that
subactivities are sent a message before their parents.

The routines found in ipc2.c use the following structure to construct lists of
activities/objects that are to receive a message:

```
typedef struct ml_node {
        char            ml_idtag[TAGLEN];
        char            ml_port[PORTLEN];
        int             ml_sent;        /* as defined in msg.h */
        int             ml_recv;
        struct ml_node *ml_next;
} MSG_LIST_NODE *msg_list, *msg_tbl[MAXLEVEL];
```

Functions:

act_cmd_send(msg, tag)
char *msg, *tag;

Send a messsage to all the subactivities of the activity whose tag is given and then send the message to the objects registered in the activity. If the msg is one of the "built-in" commands recognized by the system, update the activity's status variable. If the message is ACT_TERMINATE, notify the parent ACM with an ACM_TERMINATED message, and deregister the objects using ac_dereg_obj (see ophandler.c)

init_act_cmd(tag, level)
char *tag;
int level;

Initialize the msg_tbl array used by ac_cmd_send. This is done by performing a postorder traversal of the activity tree (child links are visited first, then right siblings) The records are read into msg_list nodes at the corresponding "level".

cleanup_act_cmd()

Free the space allocated for msg_tbl nodes by init_act_cmd

add_msgtbl(level, node)
int level;
MSG_LIST_NODE *node;

Add "node" to the msg_tbl at index "level"

add_msglist(list, node)
MSG_LIST_NODE *list, *node;

Add "node" to the message list pointed to by "list"

init_msgnode(node, idtag, port)
MSG_LIST_NODE *node;
char *idtag, *port;

Initialize "node":
node->ml_next <- NULL
node->ml_idtag <- idtag
node->ml_port <- port
node->ml_sent <- 0
node->ml_recv <- 0

Note: ml_sent and ml_recv refer to message values contained in msg.h

init_msgtbl()
Initialize the msg_tbl structure for use by act_cmd_send();

ckreplies_updstatus(actrec, new_act, msgtype)
        ACTREC *actrec, *new_act;
        unsigned msgtype;

Check the replies returned by all subactivities
to see if it matches the activity command sent.
If so, the status variable of the subactivity is updated
In addition, if the msgtype is ACT_TERMINATE, objects are
deregistered from their activities.

mk_objlist_msg(tag)
Create a msg_list for the objects contained in the activity's object list.
Used by act_cmd_send().

mk_actlist_msg(id)
create a msg_list for the activities contained in the object's activity list.
Not presently used.

:: ipc.c

scription:
er-process communication routines called by ipc2.c routines;
s tcp/ip sockets.

nctions:

socket(type, addr)
    int type;
    char *addr;

Get a socket of the given type and return result in addr. Types are "TEMP" for temporary, "PERM" for permnent sockets that are associated with an object implementor/manager or ACM, and "SERV" for sockets that are und to a server ("well-known" addresses).

If type = TEMP we don't need to bind an address to it, and therefore we don't really care about the return lue of addr. For type = SERV, we will be told what port to use as part of addr. Bind this port to the socket after creation. For type = PERM, we need to know what port number was chosen as a return value, so start at ORT_START and try to bind an address to the socket. Set addr = "port@machine" so that we know where to ake connection requests in the future

eturns:
    the result of s = socket(...)
    the port portion of address in addr for type=PERM

ndwr(dest, msg, reply, timeout)
    char *dest, *msg, *reply;
    unsigned timeout;

end a message "msg" to destination "dest" and wait for a reply within
timeout" seconds. Destination is in the form "port@node_name".

eturns:
    0: Successful operation (also returns result from dest in "reply")
    -1: Unable to connect or write to receiver

ndnr(dest, msg, timeout)
    char *dest, *msg;
    unsigned timeout;

end a message to destination - do not wait for a reply
estination is in the form "port@node_name"

eturns:
    0: Successful operation
    -1: Unable to write to receiver

reply_to(s, msg)
char *msg;

Send a reply via a socket that is already connected to the receiver

Returns:
0: Successful operation
-1: Unable to write to receiver

sendallwr(destlist, msg, reply, timeout)
char *destlist, *msg, *reply;
unsigned timeout;

send a message to the destinations and wait for a reply within timeout secs
destinations are contained in destlist and are of the form "port@machine"

Returns:
0: Successful operation
-1: failed

sendallnr(destlist, msg)
char *destlist, *msg

send a message to the destinations; do not wait for a reply.
destinations are contained in destlist and are of the form "port@machine"

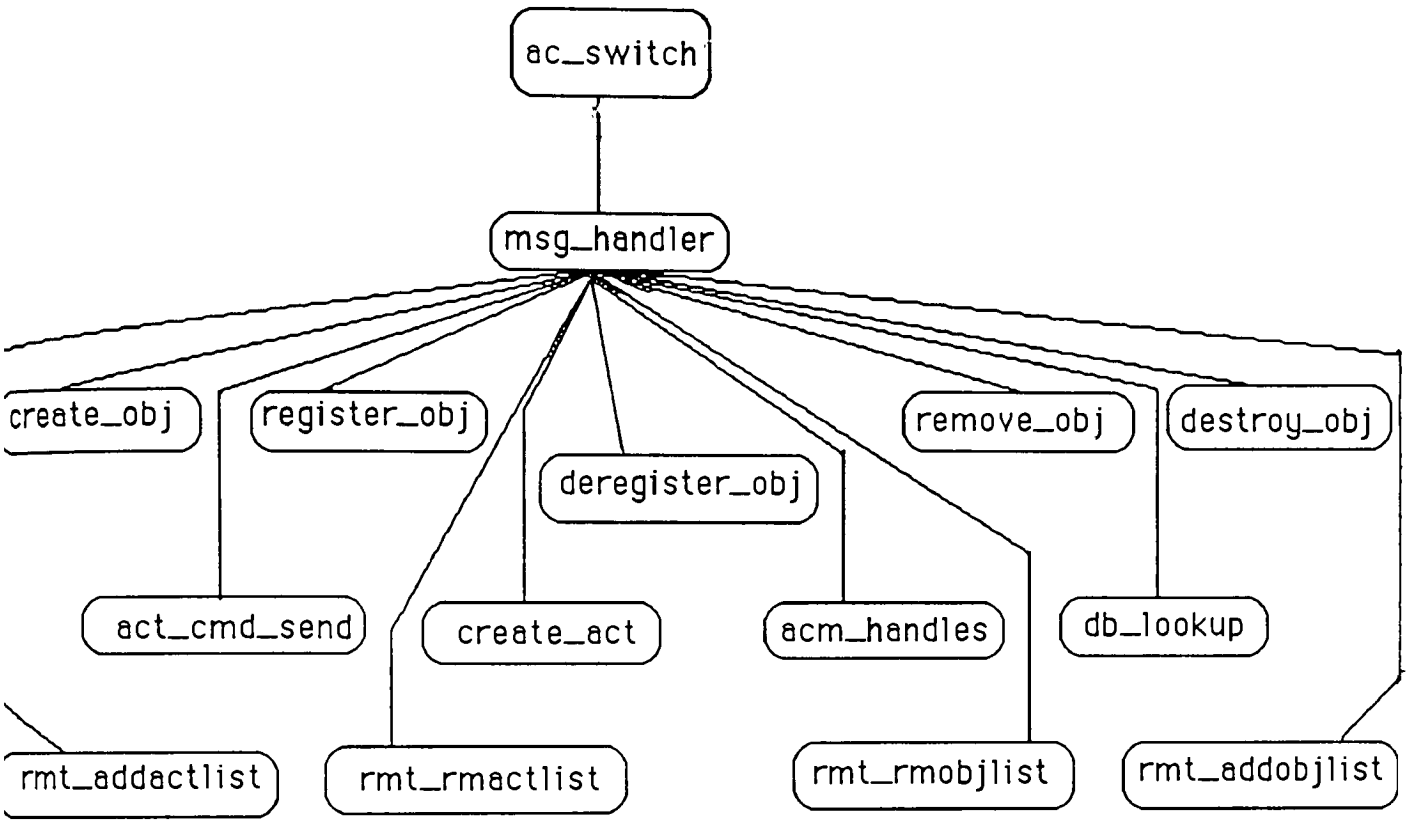
Returns:
0: Successful operation
-1: Failed (unable to write to receivers)

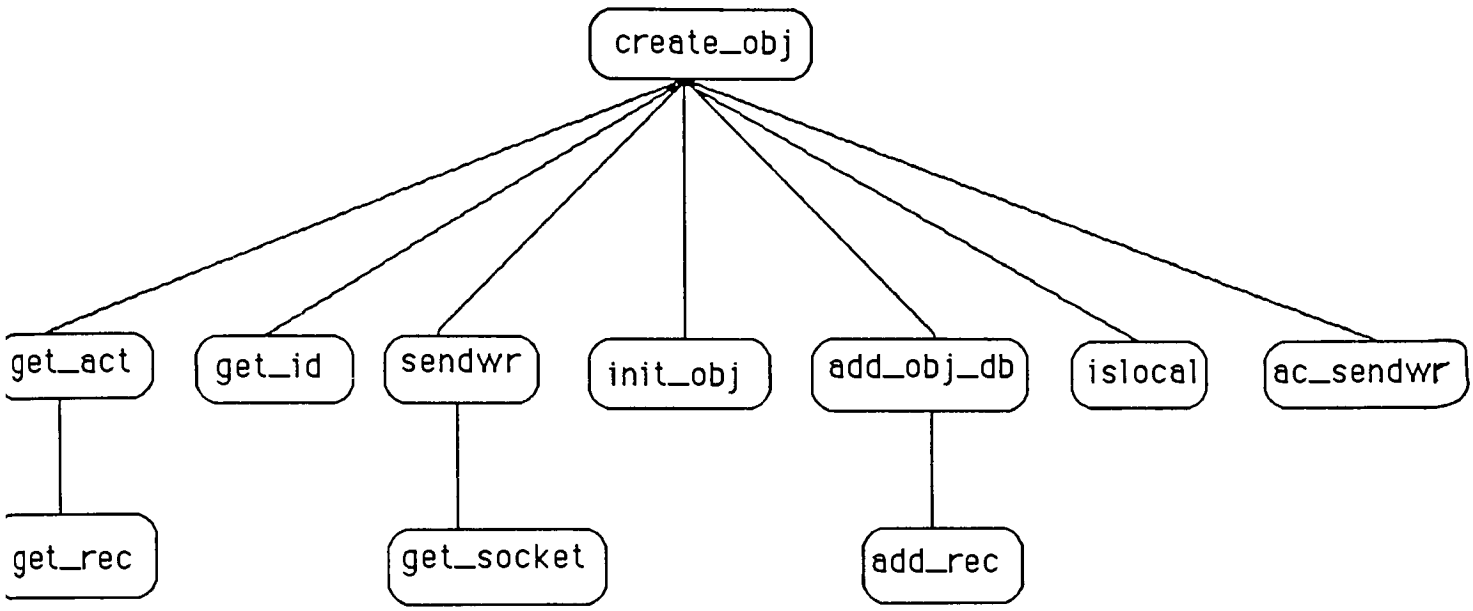
ac_sendwr(dest, desttype, msg, reply, timeout)
char *dest, *msg, *reply, desttype;
unsigned timeout;

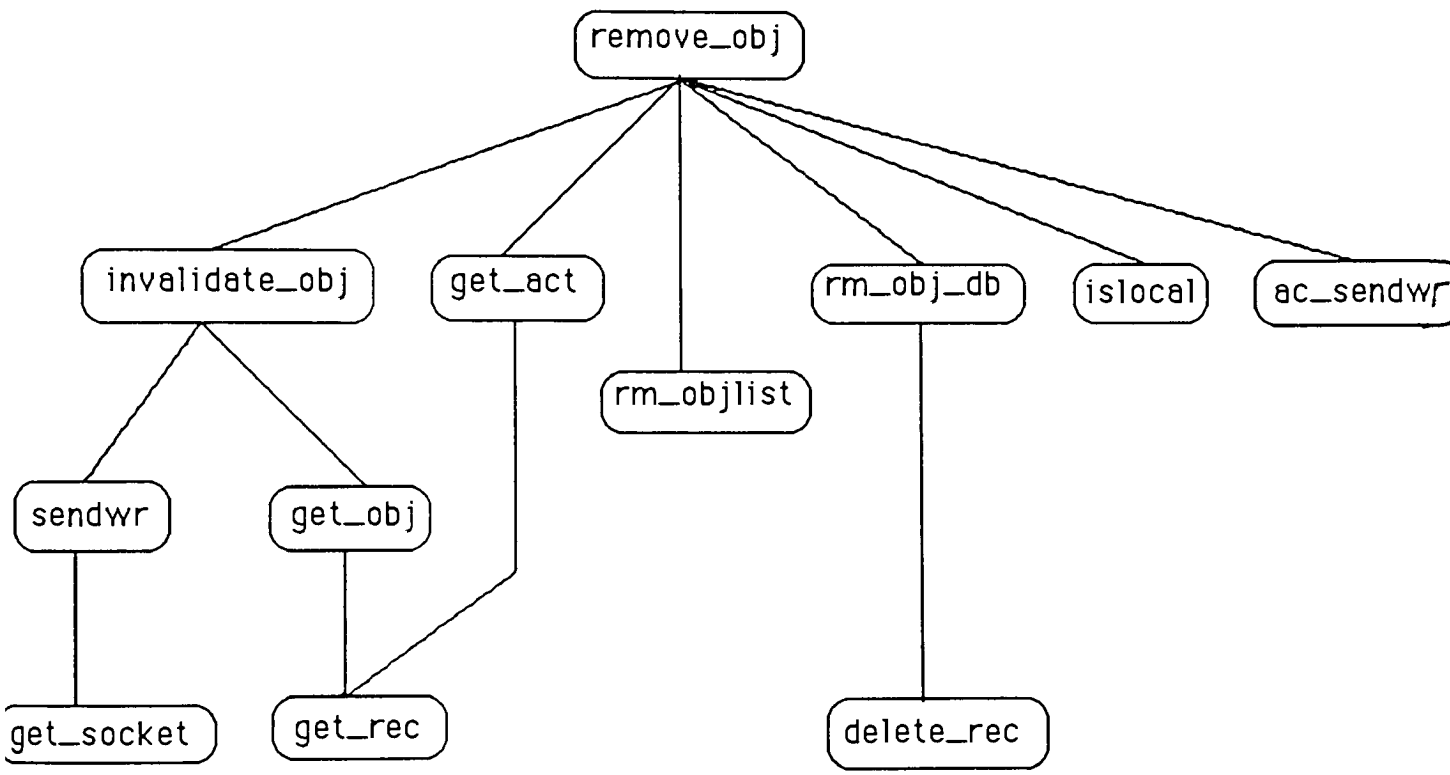
Send a message "msg" to destination "dest" and wait for a reply within
"timeout" seconds. Similar to sendwr except that the request
originates from a remote AC server. Also, destination is either
an activity tag or an object id. The flag "desttype" is
used to determine whether a tag('T') or id('I') is used.

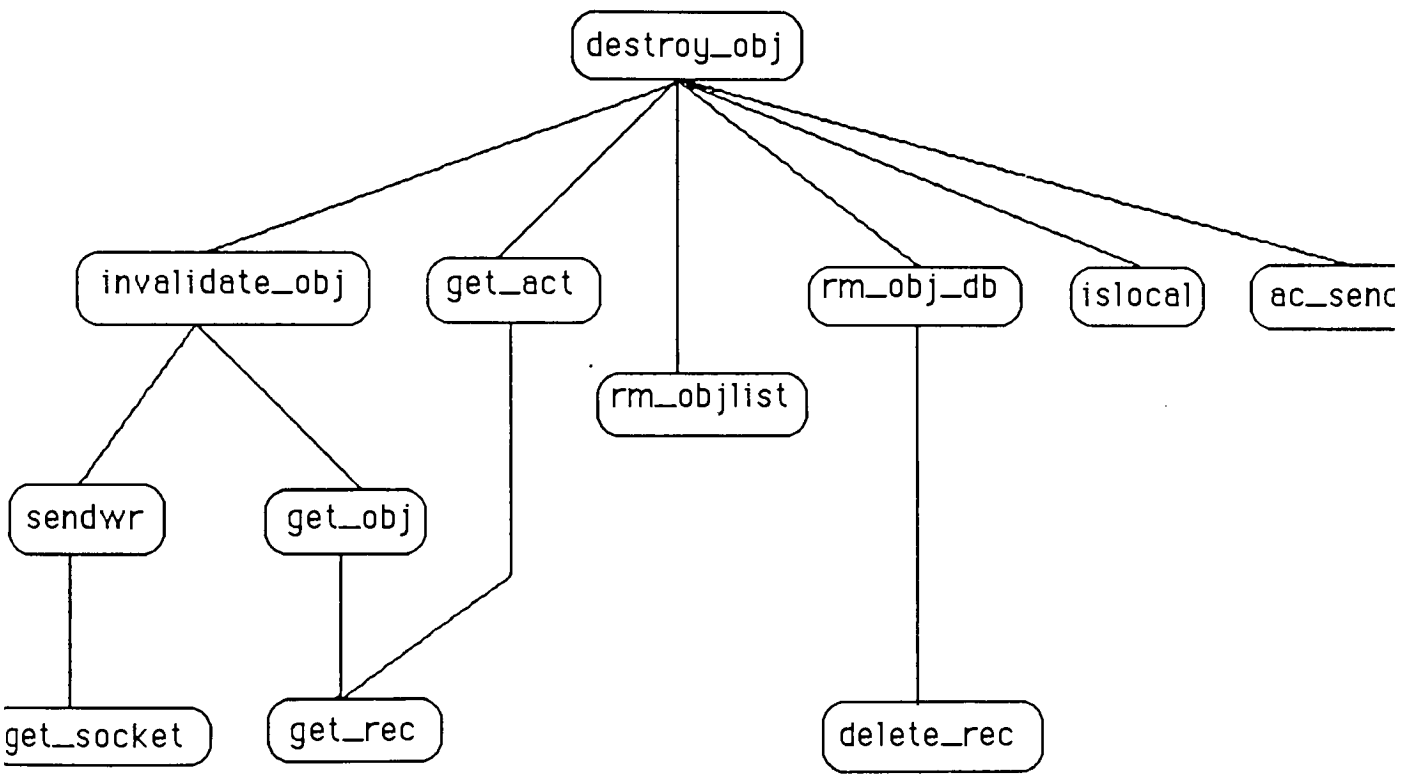
Appendix B

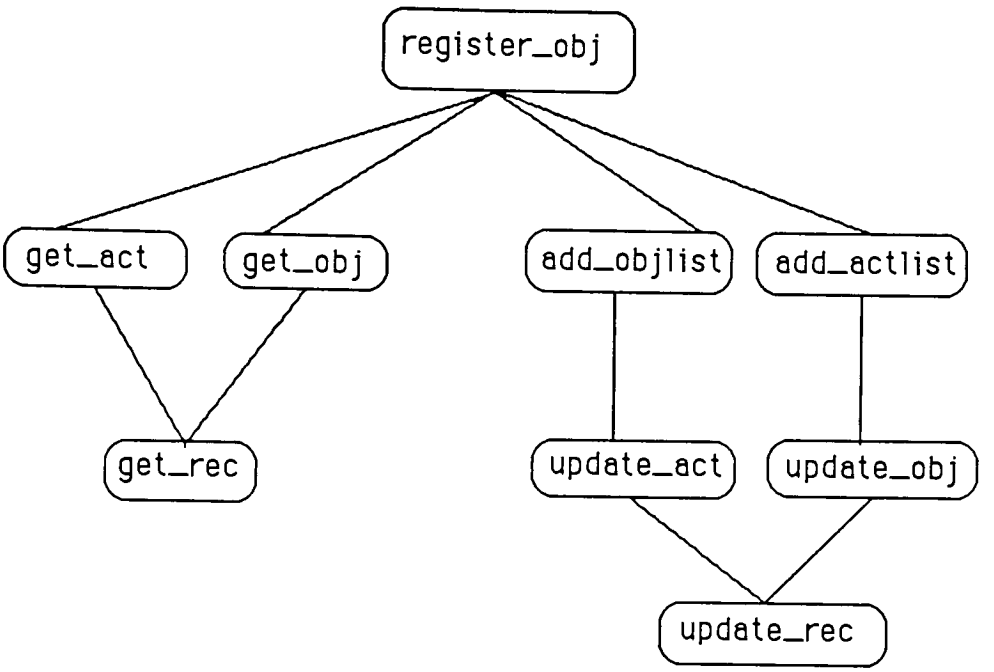
Activity Coordinator Function Diagrams

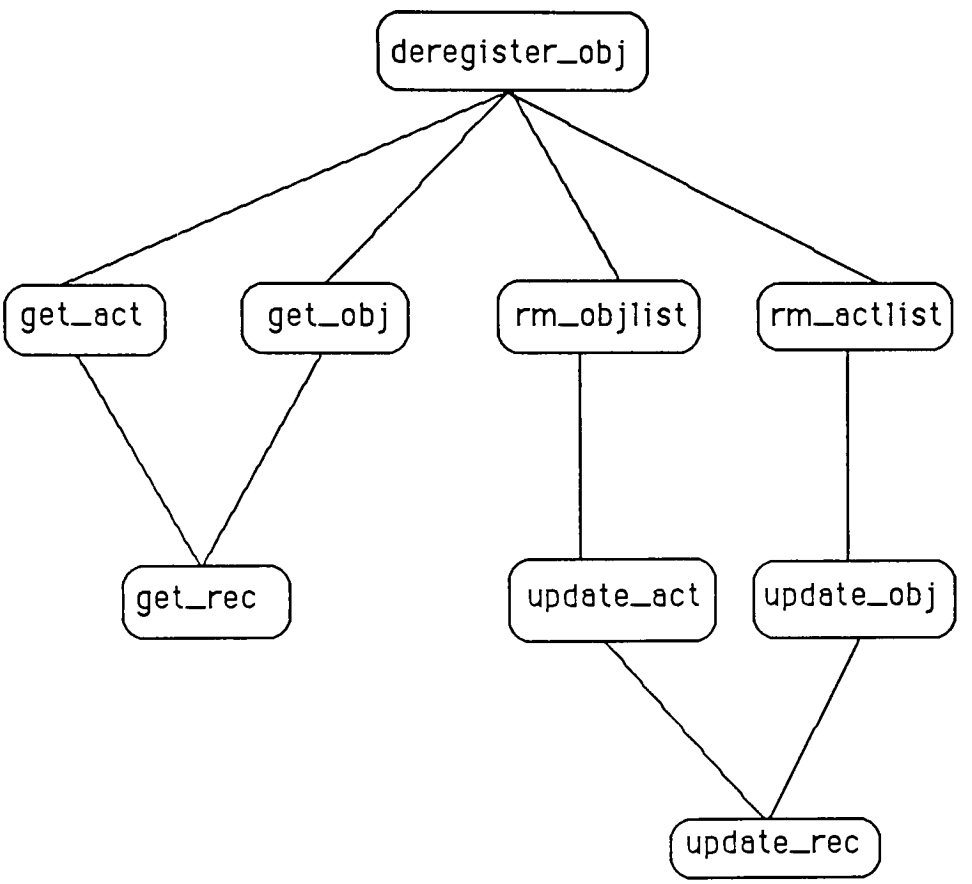

ac_switch
msg_handler
create_obj
register_obj
deregister_obj
remove_obj
destroy_obj
act_cmd_send
create_act
acm_handles
db_lookup
rmt_addactlist
rmt_rmactlist
rmt_rmobjlist
rmt_addobjlist

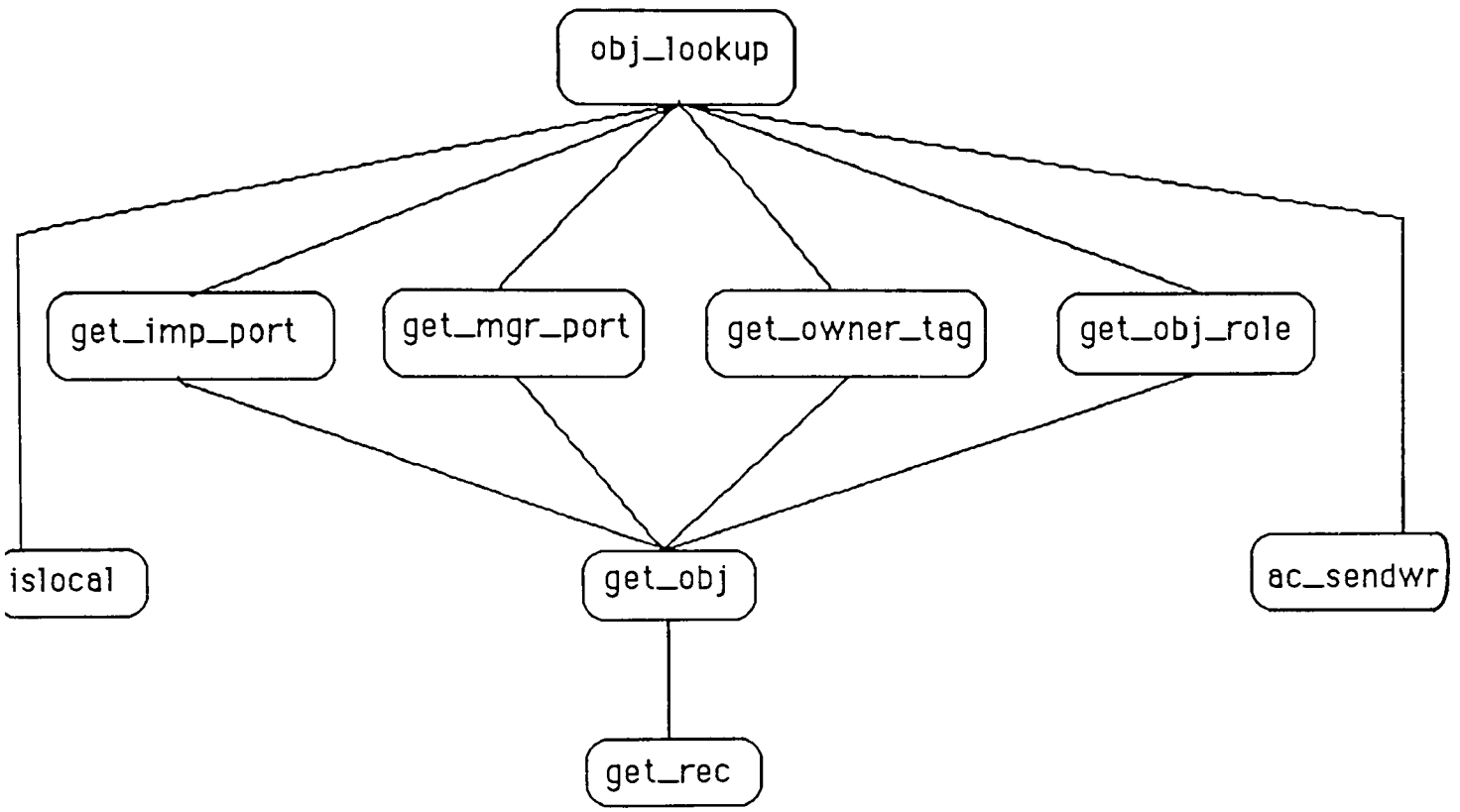

create_obj
get_act
get_id
sendwr
init_obj
add_obj_db
islocal
ac_sendwr
get_rec
get_socket
add_rec

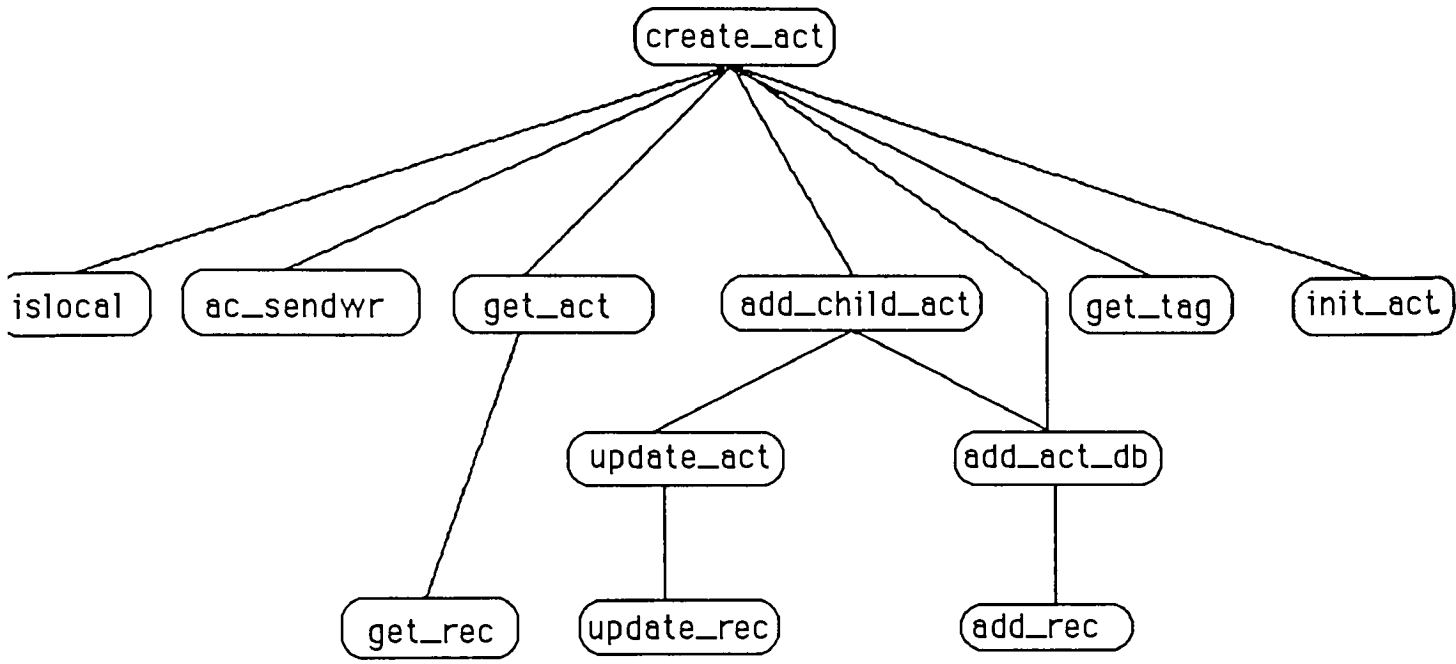

remove_obj
invalidate_obj
get_act
rm_objlist
rm_obj_db
islocal
ac_sendwr
sendwr
get_obj
get_socket
get_rec
delete_rec

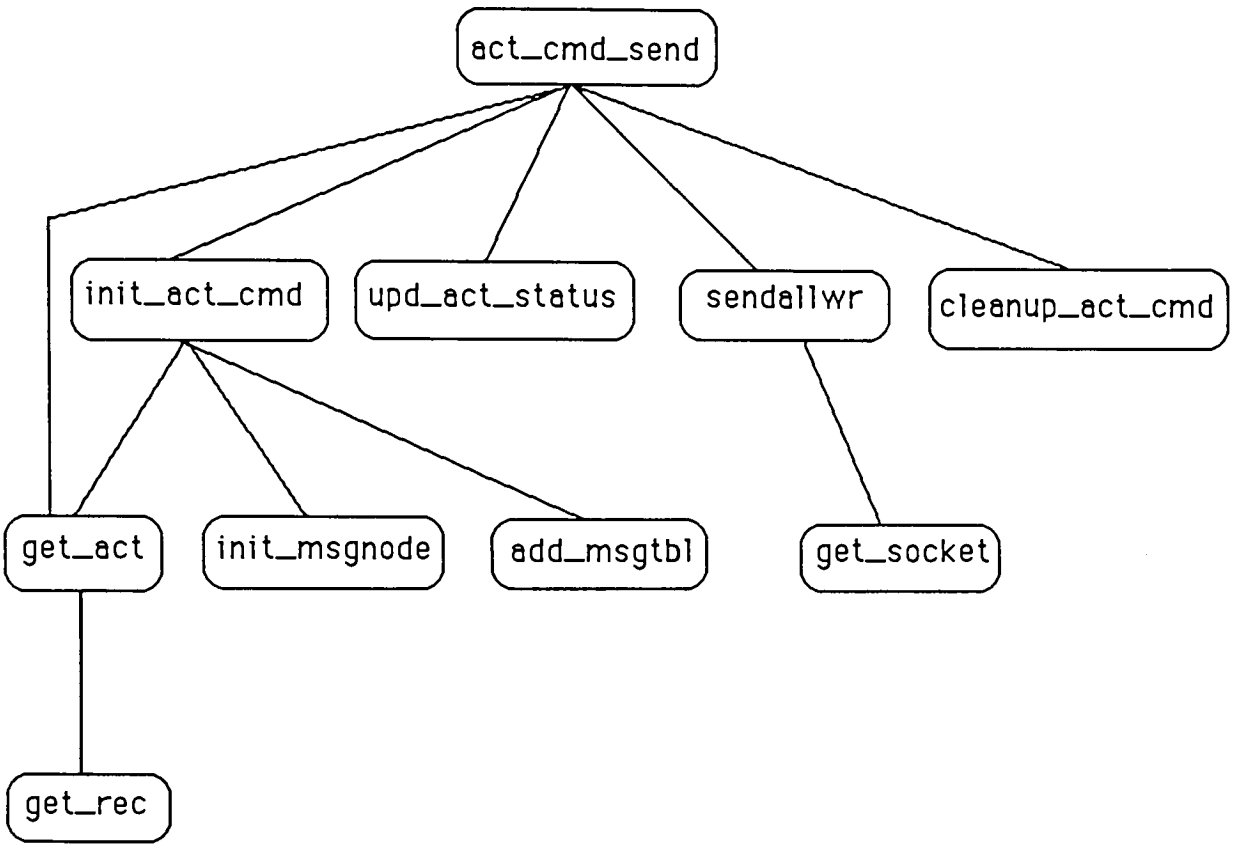

destroy_obj
invalidate_obj
get_act
rm_obj_db
islocal
ac_senc
rm_objlist
sendwr
get_obj
get_socket
get_rec
delete_rec


register_obj
get_act
get_obj
add_objlist
add_actlist
get_rec
update_act
update_obj
update_rec


deregister_obj
get_act
get_obj
rm_objlist
rm_actlist
get_rec
update_act
update_obj
update_rec


obj_lookup
get_imp_port
get_mgr_port
get_owner_tag
get_obj_role
islocal
get_obj
ac_sendwr
get_rec


create_act
islocal
ac_sendwr
get_act
add_child_act
get_tag
init_act
update_act
add_act_db
get_rec
update_rec
add_rec


act_cmd_send
init_act_cmd
upd_act_status
sendallwr
cleanup_act_cmd
get_act
init_msgnode
add_msgtbl
get_socket
get_rec