Structuring Resilient Distributed Programs with the Activity Model

Carla S. Ellis and James E. Heliotis*  
Department of Computer Science  
The University of Rochester  
Rochester, NY 14627  

TR 163  
June 1985

Abstract

Although the technology exists to build networks of computers providing a wide range of capabilities that one would like to be able to exploit in solving a problem, programming systems that facilitate the design of distributed programs are rare. This is especially true with regard to the structuring of distributed computations. Thus the focus of the work described in this paper has been to address this question of how to structure a distributed program. The result is the development of a model of activities. A second objective of the project has been to investigate strategies for managing the dynamic behavior of a distributed computation and to provide flexible tools to support implementation of such control. We propose a set of high level language constructs and a support system to allow a designer to specify responses to failures that are appropriate to the particular application under development.

*The second author's current address is: School of Computer Science and Technology, Rochester Institute of Technology, Rochester, New York, 14623.

This research was supported in part by the National Science Foundation under Grants MCS-8203306.
1. Introduction

Although the technology exists to build networks of computers providing a wide range of capabilities that one would like to be able to exploit in solving a problem, programming systems that facilitate the design of distributed programs are rare. The basic ideas about the programming methodology underlying the few such systems around are still evolving. This is especially true with regard to the structuring of distributed computations. Thus the focus of the work described in this paper has been to address this question of how to structure a distributed program. The result is the development of a model of activities.

A second objective of the project has been to investigate strategies for managing the dynamic behavior of a distributed computation and to provide flexible tools to support implementation of such control. Whenever components of a system can independently fail, it becomes important to be able to provide features such as recovery, graceful degradation of the application, or continued availability of information through replication. We propose a set of high level language constructs and a support system to allow a designer to specify responses to failures that are appropriate to the particular application under development.

Before considering the elements needed in a model for organizing distributed computations, it is valuable to look at the characteristics of various applications that make good use of distribution. In order for it to make sense to design a distributed solution for a problem, there must be some benefit derived from spreading code and data out among sites of a network. One strong motivation is the natural geographic dispersal of users or other sources of information. Mail systems and sensor networks are examples. Other reasons include the need to accommodate growth beyond the limits of a single machine and a desire for greater reliability and availability. The most studied category of distributed applications involves the storage of data (e.g. database systems and network file systems). A primary requirement for these systems is some technique (e.g. atomic transactions [Lampson 79]) for maintaining the consistency of the data entrusted to them. Another potentially large class of distributed programs which has not received as much attention can be represented by process control systems. The requirements of these systems are substantially different from those of the database applications. In particular, there is more emphasis on interactions with the physical world and on timing; consequently, the viable options for dealing with failures are likely to be quite different. Most of the programming systems proposed to date seem to favor the first category of applications. The structuring concepts presented here should have relevance to a wider range of problems.

The activity model grew out of early experiences with distributed programming languages at the University of Rochester, namely the PLITS project [Feldman 79]. A previous report [Ellis 82] outlined our initial ideas and some preliminary development on the activity concept. The model matured in a Ph.D. thesis by Heliotis [Heliotis 84] which is the basis for this paper.

Underlying the activity model is a recognition that there are two orthogonal kinds of structure that must be captured in a model of distributed programs: the structure of objects and of computations. Thus one dimension is satisfied by the
familiar object model [Jones 79] where an object is a module that has autonomous control over its data structures and modifies them only in response to operations specified in its interface. Activities address the other dimension. An activity is a tangible handle on the dynamic structure of a computation that may involve a number of objects. From the programmer's point of view, an activity ties together a group of objects that are contributing to a common goal. Activities can subsume the intuitive notion of a distributed job.

One of the key aspects of the model is the treatment of sharing of objects by multiple activities. This is an important issue since the shared server is so prevalent in the organization of current distributed systems. It has also proved to be a problematic one. When a shared object may participate in several activities concurrently, there are questions of how to manage whatever state must be kept inside of the object for each activity and of jurisdiction (e.g. which activity can be responsible for termination of the object). The next section gives our answers to these questions.

A major advantage of the activity concept is that it provides a context for the handling of faults that may arise in a distributed computation and for other kinds of dynamic control. As an example, 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. Activities are intended to support flexibility of management options.

In order to motivate the need for the structuring constructs we propose, consider the shortcomings in the traditional approaches to writing distributed programs. Typical ways in which such programs are organized range from a flat space of processes communicating through some form of asynchronous messages (e.g. port-based as in [Rashid 81], link-based as in [Solomon 79]) to the client-server model using remote procedure call (e.g. Courier [Xerox 81]). These various styles reflect the attention that has been paid to the communication aspect of designing distributed systems. The pattern of communication is usually the only information available in the support system during execution. Thus, notification of the death of a process must propagate through the established connections to the other processes directly or indirectly affected. This propagation may take considerable time. With each process acting from its own local perspective, no unified response is possible when a failure is detected. What is lacking is information about the logical relationships between the processes.

There is typically also a problem with the readability of the code in such systems. The code often must contain a lot of bookkeeping for detecting communication problems that gets intertwined with the code for doing the actual work. Figure 1 gives an example of such cluttered code taken from a working distributed file system. This uses a port-based IPC mechanism with the system providing notification of the disappearance of a port in the form of emergency messages to processes holding send rights to the port. Timeouts are available, but not used in this example. Instead, the death of a port is used to represent the death of its process. The goal behind this code is just to receive one normal (i.e. non-emergency) message. The emergency
while (TRUE)
{
    while (msg_receive(ports, msg, INFINITE_WAIT) != MSG_SUCCESS)
    {
        if (errno == EINTR) /* should never be true. */
            log_problem(FATAL, "receive error: %s. Bye!",
                        msg_error_message(errno));
    }
    if (msg_get_class(msg) == MSG_EMERGENCY_MSG)
    {
        (*emergency_msg_handler)(msg);
        msg_port_release(msg);
    }
    else /* finally - a good normal message */
        break;
}
/* ------------------ GDS_emergency_msg_handler ------- */
/* (Almost) all emergency messages come here. This routine is called by
"receive", among others. Checks to see if our user has died, and,
if so, gives up. If it's a server, take it out of the up list.
If something else, do appropriate default processing and hope it was
nothing important. */

GDS_emergency_msg_handler(msg)

    msg_handle msg;
    
    switch(msg_get_id(msg))
    {
    case MSG_PORT_DELETED:
        { int dead_port;
            (void) msg_scanf(msg, "d", &dead_port);
            if (dead_port == user_port)
                log_problem(FATAL, "User port went away!");
            else /* assume it's a server port. */
                delete_server(dead_port);
            msg_release_port(dead_port);
            break;
        }
    default:
        default_emergency_msg_handler(msg);
        break;
    }
}

Figure 1 Traditional Style Code
message handler routine shown uses knowledge of the application to deal with the problem if it can and otherwise passes it off to a default handler (not shown). Often with this approach, a port to a process is acquired solely for the purpose of receiving notification should that process die. Actually, attention has been paid to structuring this code as well as the programming system allows, but the important point is that procedures devoted to error handling have syntactically identical status as the procedures doing productive work.

In contrast to the primarily communication oriented systems described above, there are a few other research efforts in which structuring plays a central role. Most of these systems are based on some variation of a model of objects and (nested) atomic actions. These include the Argus project at MIT [Liskov 83 a and b, Weihl 85], the Clouds/Aeolus project at Georgia Tech [LeBlanc 85 a], and others [Shrivastava 81, Jensen 82, Schwarz 84, Spector 83]. It is significant that the object/action models capture the two dimensions of distributed system structure that we have identified. However as suggested earlier, the emphasis on atomicity as the central theme in managing the computation is not appropriate for all possible applications. Recent work has acknowledged this in some sense by providing more flexibility while remaining within their basic framework [Weihl 85, Schwarz 84]. The Clouds/Aeolus approach appears closest in philosophy to that of activities by allowing the programmer to specify application specific recovery and synchronization, even permitting breaches to serializability when semantically appropriate.

Another proposal for structuring distributed programs is the HPC model [LeBlanc 85b] that emphasizes the functional composition of objects from other objects. In this one dimensional model, the notion of activity is represented as a composite object. The model does an excellent job of supporting abstraction, but important issues related to sharing and control must still be worked out.

In the next section, the definitions of the activity model are presented and features of the support system are discussed. Section 3 describes the most important of the language constructs developed to provide a high level interface to the activity system. Then in section 4, an example is given to illustrate the use of the model and language constructs in developing a distributed program.

2. The Activity Model

Activities can be defined from several different perspectives. The informal description presented earlier of an activity being a collection of objects that are all cooperating in some way towards a common goal is vague but still valuable in providing the user with the right intuitions needed to govern its use. Some of the objects may be shared. Thus if one could take a snapshot of an activity based system, the activities would be represented as intersecting set relationships imposed upon the objects existing at that point in time.

This view is incomplete in that it does not capture the notion of subactivities. The model allows for a hierarchy of activities. The objects participating in one subactivity can be insulated from problems occurring in other logical parts of the
computation. The need for an atomic step within a long-lived computation is another good reason to create a subactivity. When atomicity is required, subactivities resemble nested atomic actions. Thus, an activity can be defined as a tree, where all the internal nodes represent subactivities and the leaves are the objects which participate in their parent node activities. The idea of shared objects corrupts the tree at the leaf level so that a more accurate term for the structure would be a directed acyclic graph.

A more concrete definition of an activity is a dynamic identifiable collection of state information that is spread out among the constantly changing set of objects in a computational network. Each object holds only part of the state associated with an activity and sharable objects must maintain multiple sets of such data. In order to satisfy the need to identify items belonging to an activity, explicit tagging becomes an important aspect of the paradigm. Each activity has a unique identifier, its activity tag, that serves as the tangible handle to that activity. Messages and data structures inside objects can carry the tag of the related activity.

Finally, an activity is a context for runtime management and control. Control over an activity as a whole is realized by propagating activity commands to all of its components (i.e. its subactivities and the objects participating in the activity). Some examples of commands are terminate, suspend, trace communication, and the atomic transaction primitives of the two phase commit protocol. Note that these are commands to be applied to activities not individual objects. To clarify this point, consider how an object that is shared by multiple activities might respond when it receives a command to suspend one of them. It may choose to avoid doing work on behalf of the suspended activity, but it probably should not suspend its own execution. For commands that are built into the activity system, there are default responses defined which objects can employ. However, it seems important that a programmer have the freedom to change the semantics of a command for his own objects or activities. Thus, we provide a mechanism whereby copies of these commands are sent to user supplied code for interpretation. Tools for specifying command handlers are described in section 3.

Objects may be either active (e.g. modules) or passive (e.g. files). In either case, there exists an active object associated with the object (possibly itself) that plays the role of its manager. It is this object manager that provides the interpretation of incoming activity commands for the object. Each type of object also has an object implementor responsible for creation and destruction. For passive objects, the implementor also performs the actual operations on the object and serves as the object manager.

An activity control module (ACM) is a special component with the designated responsibility to represent the activity. Just as an object manager may supply an alternative interpretation of an activity command for its object, the ACM may do so for the entire activity or for any of its subactivities or objects. ACMs are also notified about the death of any object participating in their activities.

Exactly one activity, the owning activity, is responsible for each object even though it may be shared. The ACM for an object's owning activity has jurisdiction
over the object.

The foundation of the activity system is the activity coordinator which keeps track of all the activities in the network. In other words, it maintains a data base of activity trees. An object is said to be registered within an activity when an entry is created for it as a direct descendent of the indicated activity node. The coordinator also acts as the clearinghouse for propagating activity commands, routing requests for certain object manipulations, and forwarding emergency notices. The interactions between the activity coordinator, object managers and implementors, and ACMs are evident in the description of how the coordinator performs the various clearinghouse functions.

First consider the protocols for propagating activity commands. A program may send a request to the activity coordinator that an activity command be applied to some activity. The coordinator searches the entire activity tree rooted at the activity mentioned in the request in order to find all the objects and subactivities that should receive notice of the command. For any node in the tree representing an activity, the command is first recursively executed for its subactivities, then the object managers for objects registered in that activity receive the notice, and finally, the ACM of the activity itself receives the command. The ACM may effectively receive the notice at additional earlier points in the protocol by pre-arranging with the coordinator that it wants to receive the command in lieu of one or more of its subactivities or objects. In this way, ACMs can provide local interpretations of commands. For certain built-in commands, there are default actions that are taken for activities with no user-supplied ACM.

Commands requesting object creation or destruction and those involving the registration of objects within activities form the base of a group of system primitives called object commands. These commands are sent to the coordinator which passes the request on to the appropriate implementor and/or manager. These commands do not propagate through the activity tree since they concern a single object and a single activity.

There is a final category of notices called emergency notices that inform interested parties of the death of an object. The information flows from the implementor of the object to the activity coordinator and from there to the ACMs of all the activities in which the object was registered. The failure of an ACM is currently treated as a failure for the entire activity.

3. Programming Language Features

Given the incorporation of an activity coordinator into the network operating system, the activity model could be used to design programs which would then be expressed at the relatively low level of system calls embedded in a standard implementation language. This would allow access to the coordinator's services of registration and notification of activity-related events, but the code would be hard to read and maintain. The problems of bookkeeping clutter seen in traditional distributed programs would not be alleviated. In order for the activity model and support
system to have an impact on the development of distributed systems, a high level programming language is needed.

Rather than designing a new language from scratch, we built upon the PLITS effort and concentrated on just those features that could be added to a reasonably powerful message-based programming language to make activities usable. We chose Mesa [Mitchell 79] augmented with PLITS constructs as the base language. Although it was not as well known as other languages, Mesa offered an attractive exception handling paradigm that could be expanded to handle other kinds of notifications. The contribution from PLITS was primarily the notion of asynchronous messages consisting of (named-slot ~ value) pairs.

Although a full language design is presented in [Heliotis 84], the language constructs introduced do not depend intimately on the particular base language chosen. The ideas behind the constructs should carry over to many other languages. The list of features breaks up into a few categories, each of which addresses a different area of concern. Thus, there are syntactic constructs to distinguish between the few basic kinds of modules known to the coordinator (ACMs, self-managing objects, and object implementors). The organization of the code for a module is aimed at reducing the problem of cluttered code. Finally, there are features to aid in intelligently exploiting activity tags and to invoke the services of the coordinator.

The only object type provided by the system is the module, a combination of code and data with a message-based interface to the outside world. All other object types are essentially passive and implemented by a user-written module that acts as the object implementor for the type. The activity coordinator gives different treatment to certain modules that perform special functions with respect to activity management. Ordinary modules become the active objects in a computation. Keywords declare whether an object serves as implementor for a passive type and whether it is self-managing. This information affects the registration of the object. Such modules should include sections of code for handling the object and activity commands that may arrive as a result. ACMs are special modules not considered objects by the coordinator and contain additional sections of code for describing responses to various kinds of notices that only come to ACMs. These include the emergency notices, user-defined notices, and activity notices sent to the ACM after the command has successfully propagated to everything else in the activity.

The bookkeeping and exception handling code is syntactically separated from the normal code in special notice catcher sections. Object implementors and managers have notice catchers corresponding to activity commands and object commands. ACMs have a section for emergency notices and two sections for handling activity commands since they may receive them at two different points in the propagation protocol: first, as a substitute for a subordinate (ACTIVITY NOTICE section) and then, after the subtree has processed the command (ACM NOTICE section).

The implementation of notice catchers creates a problem because of the inherent asynchrony these notices have with the execution of the main code of the module. There may be places in the code that should be viewed as critical regions with respect to the notice handlers. In [Heliotis 84], the solution is to define clean
points in the language that are natural boundaries for such critical regions. An alternative, tailored to Mesa and other languages with monitors, is to create a separate process within the module to deal with the incoming notice. Access to any data that must be shared with the main process takes place through a monitor.

The activity tags and object ids generated by the support system are available within the language as the new data types, ACTIVITY and ID. There is also a data type called CONNECTION used for identifiers of communication channels and many of the features designed for tags and ids apply to connections as well.

One of the most important purposes of tags is to establish an activity context. Although module objects may be shared by multiple activities, many of the statements of the language make sense only when applied with respect to a single activity. An obvious example is a statement that creates a new subactivity. The activity context identifies the one activity currently in focus for the object. This context is controlled by various statements that allow the programmer to delimit a block of code and ensure a uniform activity context during execution of those statements. The following constructs are examples:

FROM WITHIN a BEGIN
<statements>
END

This causes the statements in the block to be executed in the context specified by the activity tag, a.

UPON RECEIPT OF msg BEGIN
<statements>
END

This is a form of receiving a message that sets the activity context for the body of the block to the tag of the message received, msg.ACT.

FOR EACH a : ACTIVITY TAG DO
<statements>
ENDLOOP

This generates the tag of every activity in which the object is registered and reestablishes activity context for each iteration of the loop. The clean points for executing notice handlers occur immediately before statements that are capable of establishing a new activity context.

Shared objects need some means to help organize their internal data structures based on relationships between the data and the activities in which the objects are registered. In addition, user-written object implementors need to organize their state according to the object instances they represent. The second major use of activity tags and object ids is that of maintaining tagged variables. The language includes dynamically sized arrays indexed by these tags. To illustrate how this construct can be used, imagine that an object maintains a FIFO queue of orders from multiple activities and that it receives notification that one of those activities is terminating. The tags on entries in the queue allow that object to selectively clean up its data structure, eliminating elements associated with the dead activity. Mentioning the
name of an activity-tagged array in an expression without appending an explicit index automatically assumes the activity of context.

This section has only highlighted some of the more interesting features added to the base language in order to integrate it into an activity-based environment. However, since many of the remaining statements are relatively self-explanatory, given an understanding of the operation of the underlying system, the following example should be readable in spite of the omissions.

4. An Example

The benefits of the activity model are most apparent in large complex systems, which makes finding an interesting small example a difficult task. Thus the problem we present is artificial and oversimplified. It is intended to illustrate the use of activities to express a situation of continued but degraded service when certain failures occur.

The application is the operation of an assembly line for constructing widgets. The earlier stages in manufacturing these products is time-consuming compared with the later operations. Consequently, an effective configuration may have two parallel conveyor belts that eventually merge for the faster final construction steps. Along each belt are some number of robots performing tasks and one at the junction merging the two streams. A breakdown somewhere on one of the branches eventually shuts off that stream, leaving a simple pipeline still producing finished widgets but at half the normal rate. Figure 2 shows a minimal system fitting this description.

Figure 2 Assembly Line Problem
In terms of the activity model, this specification of the problem has a natural expression. The robots and conveyors are the objects of the model and are represented by device driver modules. Actually, since there is probably a human operator playing some miniscule role in the system, there is an additional user module. The overall operation of the assembly line is a (hopefully) long-lived activity. The operation of each branch is modeled as a subactivity for failure containment purposes. Thus the activity structure corresponding to figure 2 is given in figure 3. The objects associated solely with the parallel belts are owned by the subactivities. The rest of the objects are owned by the higher level activity.

In order to understand how this activity structure affects the dynamic management of the computation, consider the following scenarios:

In the first case, suppose the operator wants an orderly shut down of the plant and issues a terminate activity command. The appropriate response is to flush partially completed widgets through the assembly process. The propagation protocol for the activity command does exactly the right thing in this situation. The bottom up execution of the command will allow the assembly line to be cleared. When the top level ACM receives the terminate notice in the ACM NOTICE catcher, the ACM (and therefore the activity) may die.
As a second example, consider what needs to happen if robot1 should fail. The failure is detected by the system and, through the activity coordinator, the ACM for the owning subactivity receives an emergency notice about the death of its object. This results in termination of that subactivity. During the propagation protocol, the remaining objects (i.e. the conveyor and, in a larger configuration than shown in figure 2, other robots) are given the opportunity to clean up what they can of the unfinished work. When the death of the subactivity is reported to the parent activity, its ACM responds by slowing down its conveyor and having the robot in charge of merging ignore the broken line.

We present the Mesa/PLITS/Activity code for the top level activity's ACM below:

Assembly Line Activity

--Definitions Module "ALDefs"

ALDefs: DEFINITIONS =
BEGIN
--some message slots
    call:{slowdown, clear, Astopped, Bstopped} FOR MESSAGE;
   neighbor: ID OF MODULE FOR MESSAGE;
   clearing: BOOLEAN FOR MESSAGE;
--modules
   AssemblyLine: ACTIVITY CONTROL MODULE;
   ConveyorC: ROLE CODE MODULE;
   Branch: ACTIVITY CONTROL MODULE;
   User: ROLE CODE MODULE;
   Robot3: ROLE CODE MODULE;
   Robot4: ROLE CODE MODULE;
END. --Definitions
Activity Control Module "Assembly Line"

DIRECTORY ALDefs;
AssemblyLine: ACTIVITY CONTROL MODULE ==
BEGIN
OPEN ALDefs;
EMERGENCY NOTICES BEGIN
    Robot3 \( [r: ID \text{ OF \text{MODULE}}] \Rightarrow \) BEGIN
    --flush assembly line downstream by passing "clear" token
    SEND (call "clear") TO ConveyorConn;
    END;
    Robot4 \( [r: ID \text{ OF \text{MODULE}}] \Rightarrow \) BEGIN
    --if last robot fails, just let everything die.
    TERMINATE[];
    END;
    ConveyorC \( [c: ID \text{ OF \text{MODULE}}] \Rightarrow \) BEGIN
    --terminate upstream, clear downstream
    DESTROY[robot3];
    SEND (call "clear") TO Robot4Conn;
    TERMINATE[] ABOUT branchA;
    TERMINATE[] ABOUT branchB;
    END;
    User \( [u: ID \text{ OF \text{MODULE}}] \Rightarrow \) BEGIN
    --we really don't care about the human
    END;
    OTHER \( [a: ACTIVITY] \Rightarrow \) BEGIN
    --DYING notice for subactivity
    SEND (call "slowdown") TO ConveyorConn;
    IF a = branchA
    THEN SEND (call "Astopped") TO Robot3Conn;
    ELSE SEND (call "Bstopped") TO Robot3Conn;
    END;
    END;
ACM NOTICES BEGIN
    TERMINATE => BEGIN
    DESTROY [Self];
    END;
USER NOTICES \( [msg: MESSAGE] => \) BEGIN
    --When the "clearing" token arrives at robot4, it will
    --send me a message telling me that flushing is done
    TERMINATE[];
    END;

-Definitions
    robot3, robot4, conveyorC, user: ID \text{ OF \text{MODULE}};
    Robot3Conn, Robot4Conn, ConveyorConn, UserConn: \text{CONNECTION} \text{ TO \text{MODULE}};
    branchA, branchB: \text{ACTIVITY};
--Start things up
[robot3, Robot3Conn] <- INstantiate Robot3;  
[robot4, Robot4Conn] <- INstantiate Robot4;  
[conveyorC, ConveyorConn] <- INstantiate ConveyorC;  
[user, UserConn] <- INstantiate User;  
branchA <- ACTIVATE Branch;  
branchB <- ACTIVATE Branch;  

--Tell everyone who their neighbor downstream is for clearing
SEND (neighbor - conveyorC) TO Robot3Conn;  
SEND (neighbor - robot4) TO ConveyorConn;  
--robot4 knows to send the user notice to the ACM

END. -- Assembly Line ACM

5. Conclusions

One of the main contributions of the activity model is that it provides a framework for thinking about how to design a distributed program that can respond to changes in its environment in a way that is appropriate to the application. The language offers a template for modules that encourages the programmer to address management issues in a more organized and possibly a more ambitious manner. The activity system has a more global view of a distributed computation than any of the component objects locally see and this allows more unified management strategies to be developed.

The status of the project is that a detailed design of a prototype activity system has been done. Plans call for implementation of a distributed activity coordinator in the near future. A complete language definition has been provided. We hope to verify that the language constructs are in fact compatible with other base languages. There are a number of additional issues that deserve consideration such as protection, techniques for allowing an activity to survive the failure of an ACM, and dynamic substitution of malfunctioning or dead modules.

Acknowledgements

We would like to thank the Department of Computer Science at Duke University for providing the facilities used in the preparation of this paper.

6. References

[Ellis 82]
[Feldman 79]

[Heliotis 84]

[Jensen 82]

[Jones 79]

[Lampson 79]
B. Lampson and H. Sturgis, "Crash Recovery in a Distributed Data Storage System", Xerox Parc, Palo Alto, CA, 1979

[LeBlanc 85a]

[LeBlanc 85b]

[Liskov 83a]

[Liskov 83b]
[Mitchell 79]

[Rashid 81]

[Schwarz 84]

[Shrivastava 81]

[Solomon 79]

[Spector 83]

[Weihl 85]

[Xerox 81]