GEORGIA INSTITUTE OF TECHNOLOGY — OFFICE OF CONTRACT ADMINISTRATION

PROJECT ADMINISTRATION DATA SHEET

[x] ORIGINAL [ ] REVISION NO. ____

Project No. G-36-645 (R-6100-0A0) GTRC/~~GIT~~ DATE 3 / 11 / 86

Project Director: R. Le Blanc School/~~XXX~~ ICS

Sponsor: USAF/RADC

Griffiss AFB, N.Y. 13441-5700

Type Agreement: F30602-86-C-0032

Award Period: From 2/18/86 To 2/17/88 (Performance) 3/17/88 (Reports)

Sponsor Amount:	This Change	Total to Date
Estimated:	$ 389,856	$ 389,856
Funded:	$ 146,196	$ 146,196

Cost Sharing Amount: $ N/A Cost Sharing No: N/A

Title: Fault Tolerant Software Technology for Distributed Computing System.

ADMINISTRATIVE DATA OCA Contact Ralph Grede X-4820

1) Sponsor Technical Contact:

Mr. Richard Metzger
USAF/RADC/COTD
Griffiss AFB, N.Y. 13441-5700

2) Sponsor Admin/Contractual Matters:

Office of Naval Research
Resident Representative
206 O'Keefe Building
Georgia Institute of Technology
Atlanta, Georgia 30332

Defense Priority Rating: DO:A7 Military Security Classification: U

(or) Company/Industrial Proprietary: N/A

RESTRICTIONS

See Attached Government Supplemental Information Sheet for Additional Requirements.

Travel: Foreign travel must have prior approval – Contact OCA in each case. Domestic travel requires sponsor approval where total will exceed greater of $500 or 125% of approved proposal budget category.

Equipment: Title vests with Sponsor –

COMMENTS:

This is an advance Payment Pool Agreement. No Foreign Nationals are to be assigned to this project without sponsor approval.

COPIES TO: SPONSOR'S I. D. NO. 02.104.000.86.005

Project Director	Procurement/EES Supply Services	GTRC
Research Administrative Network	Research Security Services	Library
Research Property Management	Reports Coordinator (OCA)	Project File
Accounting	Research Communications (2)	Other Legal

12/88

GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION

NOTICE OF PROJECT CLOSEOUT

Date 12/9/88

oject No. G-36-645 Center No. R6100-0A0

oject Director R.J. LeBlanc, Jr. School/Lab ICS

onsor Air Force

ntract/Grant No. F30602-86-C-0032 GTRC XX GIT ____

ime Contract No. N/A

tle Fault Tolerant Software Technology for Distributed Computing System

fective Completion Date 2/17/88 (Performance) 3/17/88 (Reports)

oseout Actions Required:

__ None
X Final Invoice or Copy of Last Invoice
X Final Report of Inventions and/or Subcontracts - Patent Questionnaire sent to PI
X Government Property Inventory & Related Certificate
__ Classified Material Certificate
X Release and Assignment
__ Other _____

cludes Subproject No(s). N/A

bproject Under Main Project No. _____

ntinues Project No. _____ Continued by Project No. _____

stribution:

__ Project Director
__ Administrative Network
__ Accounting
__ Procurement/GTRI Supply Services
__ Research Property Management
__ Research Security Services

X Reports Coordinator (OCA)
X GTRC
X Project File
X Contract Support Division (OCA)
___ Other _____

Contents

Part I

An action-based programming model

1 Introduction

One of the major tasks for the 1980s and 1990s is integrating data communications and computer technologies. Among the most important examples of such integration is the group of Command, Communication, Control, and Information (C^3I) systems required by many military and civilian applications.

C^3I systems perform complex missions: some functions involving human/computer interaction, and others controlling some mechanism or process. Such large systems are likely to be distributed and to incorporate such subsystems as databases, operating systems, real-time control systems, graphics programs for data acquisition and display, and various tools for monitoring, maintaining, enhancing and tuning the system. A system of such complexity may involve millions of lines of code, with individual subsystems each accounting for as much as ten percent of the total.

1.1 Action-Based Programming and C^3I

Recent research in the area of program development methodology has described new paradigms called "object-oriented" and "action-based" programming. The literature on "object-oriented" programming focuses primarily on the problems of specifying and implementing objects. The focus of this handbook is on the issue of defining operations which can manipulate those objects. We are concerned not with the functional correctness of the individual operations but with their interactions. Our use of the term "action-based" programming underscores this emphasis. An "action" is understood to be an operation which displays certain properties when interacting with other actions. In particular, actions are units of work which can be scheduled, synchonized and managed by various system services and which may exhibit concurrency and failure atomicity.

In general terms, the program state is partitioned into a number of objects. The state within the object can be examined and manipulated only by operations previously defined

for the object. Typically, the programmer's choice for objects and their associated operations reflects a high-level, conceptual understanding of how the system is supposed to operate. When coding operations as actions, a programmer must consider the operations not only from the perspective of their functional correctness but from the perspective of their interaction with other operations as well.

A programmer must be concerned with how actions may exchange information when they are executed concurrently. Typically, the criterion is that actions may execute concurrently provided the final result is equivalent to a result which may have been achieved had the executed serially i.e., the programmer is concerned with preserving the "concurrency atomicity" of the operations.

A programmer must also be concerned with the precedence relationships among actions. For example, a file must be opened before **read** and **write** operations are performed against it. The problem of ensuring that the precedence relationships among concurrently executing actions are satisfied is known as "process synchronization." Process synchronization imposes constraints on the ordering of concurrently executing atomic actions.

When fault tolerance is made a design goal, the programmer becomes concerned with preserving the "failure atomicity" of actions. To determine whether an action must exhibit failure atomicity, the programmer must examine the operations to be executed by the action. The programmer may discover that the action contains on (or several) sets of operations such that the action must either execute all of the operations in the set or none of them. Furthermore, the action must execute the operations in the set in a sequence which satisfies an precedence relationships among them.

If an action must exhibit failure atomicity with respect to a set of operations and the action fails, recovery may proceed in one of two ways: recovery may either undo the effects of those operations in the set which have already been executed, or recovery may execute, in the proper sequence, the remaining operations in the set. For example, once a file is opened it must later be closed. If an action processing a file is to display failure atomicity, then it must guarantee that, should it abort, either both the **open** and **close** operations will be performed or that open operation (and all the ensuing **reads** and **writes**) are undone.

Most of the research related to the definition of actions has been done in connection with transaction processing in database systems. This research has been aimed as providing techniques for high speed transaction processing in a distributed environment. The

requirements for such systems include a high degree of concurrency among transactions and an ability to tolerate both faulty transactions and site crashes. In database transaction processing it is permissible to satisfy these requirements by using strategies which delay the processing of some transactions, change the order in which transactions are executed, or undo all of the effects of a transaction.

A transaction may be delayed if, because of a fault or for reasons of concurrency control, a needed resource is temporarily unavailable.

If a transaction aborts its effects can be undone. It may abort because it cannot complete successfully (either because of internal errors or external conditions in the system) or it is holding resources needed by other actions. Since the effects of a transaction are confined to data stores under system control, undoing a transaction is a matter of restoring the data stores to their previous states. The aborted transaction may or may not be restarted.

In database transaction processing there are no strong precedence relationships among transactions. The delaying, aborting and restarting of transactions, may alter the resulting state of the database, but generally any of the possible resulting states will be regarded as "correct" by the database users.

C^3I systems are similar to database systems providing high speed transaction processing in that commands can be regarded as a generalized form of transaction. Commands which inquire as to the state of a particular subsystem are similar to reads from a database state, and commands which cause a subsystem to change state are similar to updates. C^3I systems have similar requirements for concurrency and fault tolerance.

Fault tolerance and concurrency control, however, are more difficult to achieve in C^3I systems. The critical difference is that C^3I systems interact with physical systems and are subject to timing and synchronization constraints.

Synchonization constraints are discussed in detail in section 2.1. That section also provides some additional discussion of failure and concurrency atomicity.

In order to satisfy timing constraints it may be necessary to adopt optimistic concurrency control methods. Optimistic concurrency control allows actions to read uncommitted data. Optimistic concurrency control is generally avoided in database transaction processing because of the need to abort optimistic readers if the action supplying them with data is aborted. In command processing it may be necessary to accept the additional cost of

aborting an action when optimistic concurrency control is used in order to avoid the delays associated with strategies in which only committed data is read.

Furthermore, the strategies used in database transaction processing to tolerate faults may not be appropriate. Delaying the execution of a command when some resources are unavailable may be inappropriate because of timing constraints. Also, it may not be possible to undo the effects of a partially executed command: the effects of the command may not be limited to data stores but may include irreversible changes to physical systems.

The concept of an irreversible change to a physical system can be illustrated in terms of a command control system for a missile. As soon as the missile is launched, the "launching action" cannot be undone. While it is possible to restore the data stores to their previous states, such a procedure would obliterate any reference to the fact that the missile had been launched. Clearly, recoverying from a faulty launch command involves subtleties not present in the simpler problem of recoverying from a faulty database update.

Recovery requires the exploitation of redundancy in data, software and hardware. Part II will how recovery mechanisms can use redundant resources to meet requirements for fault tolerance.

The four most important objectives associated with making action in embedded systems fault tolerant can be summarized as follows:

1. **Repairing an action which has aborted:** If an action faults and aborts, it may attempt to repair itself and resume executing. This is an appropriate strategy for actions which are assigned highly critical tasks.

2. **Avoiding cascading faults:** If an action faults and aborts, the recovery handler may decide to anticipate some of the problems which may ensue from that fault. It may simply attempt to ensure that failure atomicity is preserved. Furthermore, the recovery handler may send an abort signal to other actions which may have eventually needed the results of the aborted action. This strategy is appropriate if it would be costly to let the other actions proceed only to fault when they attempt to use the results which the aborted action was supposed to have produced.

3. **Aborting an action which has committed but not terminated:** Once an action has committed it is required to terminate successfully. Thus, if the main line of the action aborts, the responsibility of the recovery handler is to find an alternative way of

bringing the action to a successful conclusion. An action which executes an irreversible action can be regarded as having committed before it terminates.

4. **Avoiding cascading aborts:** Even if an action has not committed, an optimistic concurrency control strategy may have allowed another action to have read uncommitted results. If the first action subsequently aborts and undoes or modifies the uncommitted results, it will be necessary to abort the actions which were allowed to read those results. The aborted readers may not have to propagate the abort if they are able to recover (using forward recovery) in a way which does not change any uncommitted data which they may have supplied to yet other actions.

In section 2 we introduce the concept of a staged action. In Part II, we discuss how the concept of a staged action can be used to achieve these objectives.

1.2 Using the Handbook

This handbook is intended for the working programmer who wishes to familiarize himself with the use of action-based program designs in the construction of C^3I systems. In particular, this handbook considers the problem of adapting the action-based programming paradigm from the context of database transaction processing to the needs of command processing in C^3I systems. Before action-based programming can be used for C^3I systems, it is necessary to extend the conventional approaches to both concurrency control and fault tolerance. The extensions required for concurrency control and synchronization have received more attention and are better understood than the extensions required for fault tolerance and recovery. Furthermore, the problems associated with the extensions related to fault tolerance and recovery are the more subtle and difficult to solve. Thus, the major portion of this handbook is given over to a discussion of providing fault tolerance and recovery for C^3I systems.

The majority of the innovative work presented in this handbook focuses on the software-based recovery mechanisms. The software recovery techniques are presented within the framework of action-based programming. The recovery techniques described in this handbook represent a synthesis of exception handling and action-based programming. The innovations include the use of staged actions, the treatment of forward recovery as a means for preserving failure atomicity, and the use of forward recovery to coordinate the recovery

of interacting subsystems in an environment utilizing distributed control.

Exception handlers are associated with individual units of work (actions) rather than with individual units of modularity (procedures or objects). The exception handlers have access to system services not available to the mainline of the action and are used to achieve forward recovery. To emphasize these enhancements, exception handlers are termed "recovery handlers."

In this handbook we take the position that the task of providing a computation with a measure of fault tolerance is a design activity. Achieving fault tolerance requires that the system be equipped with redundant hardware, redundant sections of code and replicated data. Decisions as to which elements of the system should be backed up and how the backup elements are to be activated are design decisions. Furthermore, it may be desirable to design some elements of the primary system in ways that facilitate the provision of redundancy.

We believe the approach to recovery we describe in this handbook can be used not only to increase the reliability of a software system but also to simplify the management and maintenance of the system. For example, if actions are robust, it will be possible to bring a site down for maintenance without extensive coordination. A robust action will abort when the site goes down, and either restart when it comes up, or have made alternative arrangements in the interim. Our approach can also be used to support software maintenance and upgrades. We describe how recovery handlers can remap the code and data windows of the associated action during recovery. This mechanism provides access to backup versions of software. A similar strategy can be used to transfer control from an old version to a new version of the code for an action.

1.3 Fault tolerant actions: an overview

When designing fault tolerant actions, there are five stages of activity to consider.

1. preparing for failure by saving information which may be needed during recovery;
2. detecting that a fault has occurred or that an action has failed;
3. limiting the consequences of a failure and compensating for those consequences that could not be avoided;

4. constructing a new state for the aborted computation, including repairing data and control information and replacing faulty software or hardware; and

5. terminating recovery and restarting the main line of the computation.

Step one is carried out by the main line of the action. Step two may be done explicitly at the request of the action which is to be aborted, at the request of another action, or at the request of the run-time or operating systems. Steps three and four are carred out by the recovery handler itself. Step six is carried out by the action manager at the request of the recovery handler.

It is unreasonable to attempt to imbue every action with a maximum amount of fault tolerance. A programmer cannot anticipate every possible fault: there are too many. Furthermore, if the programmer tries to be selective and specific he will find himself guarding against errors when it would be easier to simply fix them once and for all. The programmer may also provide a measure of fault tolerance without any specialized support from the run-time or operating systems by using input validation, conditional statements directing the flow of control and explicit back tracking under program control.

When incorporating fault tolerance into actions, the programmer must consider the criticality of the action and the types of threats from which the action must be protected. A recovery handler is built only if the cost of the recovery handler is less than the expected cost of of the damage caused by the fault[1]. As part of this analysis the programmer must consider the granularity at which a particular strategy for fault tolerance should be applied. For example, one way of defending against software errors is to have two versions of a system available. If an error is encountered in the first version, the recovery mechanism automatically switches to the second. This is a strategy which is more appropriate for large-grained structures and for rather generally stated threats. It is more reasonable to have a backup version for a several thousand line section of code and for that version to be used whenever the primary section fails. The alternative is to have backups for smaller sections of code and to have those backups serve as patches for specific errors; this second approach, however, can be unwieldy if the number of threats is large.

Yet another strategy is possible and often this proves to be the most reasonable. If a subsystem, S_1, faults and is unable to provide a service to a second subsystem, S_2, then S_2

[1] i.e. expected cost = (probability of fault) * cost of fault

irreversible actions fail. They are: 1) take steps to ensure that information regarding the occurrence of an irreversible action is recorded in several locations on stable storage, and 2) make provisions for investigating the state of the system being controlled to determine whether the irreversible action in question had in fact occurred.

2 Software Designs using Actions and Objects

Our discussion of software fault tolerant mechanisms focuses primarily on the object/action model for software design as described in section 2.1. We will also discuss a hybrid model in which processes are added to the basic object/action model.

2.1 The basic model

In the object/action model, work is accomplished by executing a set of actions. The actions carry out their work by accessing and modifying the states of various objects. The objects are global to the actions and may persist even after the computation containing the actions has terminated.

An object encapsulates data structures and provides operations (called entry points) in its external interface. These operations allow actions to create objects and to observe and modify an object's state.

An entry point to an object may (1) be an action, (2) contain a set of actions, or (3) may be invoked by other actions. An action is an operation which possess certain additional nonfunctional characteristics.

Actions provide a way of structuring concurrency control, synchronization, and recovery within a computation. Actions are regarded as managable units of work and may exhibit concurrency and/or failure atomicity (actions were discussed in detail in section 1.1). By dividing a thread of control (i.e., a computation) into a set of actions it is possible to exercise a fine-grained control over its interactions with other threads.

From the perspective of concurrency control, actions represent schedulable units of work. Declaring that a sequence of statements be atomic for purposes of concurrency control guarantees that information will propagate from one statement to the next without interference from other threads.

Interference is possible if some of the variables read or modified by one action are also modified by another action. For example, if one action reads some variables before they are modified by a second action, but reads others after the second action has modified them, the interaction is not serializable. Concurrency control prevents actions from interfering in this way.

Even if two actions are declared to exhibit concurrency atomicity, it may be possible to interleave their execution, i.e., the operations within the actions may be scheduled in a way which is consistent with the view that the actions themselves executed one after the other. Bernstein's recent book [Bern] discusses methods for ensuring serializability of concurrent execution. The most commonly used technique is two-phase locking.

In database transaction processing, it is acceptable to process actions in any order. Not all sequences of actions, however, produce the same effect. For example, if for a particular account a bank processes a withdrawal before a deposit, the result may be a bounced check. Had the actions been processed in the other order, the check would not have bounced. Discussions of database transaction processing assume that the task of ordering actions is the responsibility of the user. This is not an acceptable answer in the context of C^3I systems.

A C^3I system is an example of a *command processing* system. A command processing system is an action-based system which must interface with electro/mechanical devices and humans in a manner that satisfies specified time constraints. Sequences of commands must also satisfy precedence constraints, e.g., commands which operate a device cannot be executed until after other commands have turned the device on, initialized it, and confirmed that it is operating correctly.

The command processing system must be able to synchronize the actions to ensure they are executed in the proper order. Suppose A1 and A2 are two actions. The basic possibilities are:

1. A1;A2 (A1 precedes A2),
2. A2;A1 (A2 precedes A1), and
3. parbegin A1,A2 parend (the actions may execute concurrently but the final result is either A1;A2 or A2;A1.)

Concurrency atomicity (with synchroniztion) is adequate just by itself provided actions never fail. Unfortunately, some actions may not run to completion because of problems with code, data or hardware. When an action terminates prematurely it is said to abort. Not only may actions be aborted because of internal problems, they may be aborted for reasons of systems management. If a high priority action needs the resources being used by a low

priority action, the action manager may abort the low priority action and give its resources to the high priority action. Regardless of the reason an action is aborted, something must be done to clean up after it, i.e., to recover from the abort. The problems associated with the failure of an action belonging to a set of concurrently executing and interacting actions is discussed more in section 2.2.

Internal aborts can be signaled using a mechanism for raising exceptions. If a component observes that it or another component has faulted, then the component noticing the fault can raise an exception. An exception raising facility provides a mechanism for fault recovery the fault results in system failure. Goodenough [9] has observed that exceptions may be also be used to classify results and monitor activity. In this paper we focus on signalling that a fault has been detected. When an exception is raised and a fault is indicated, an action aborts. We will emphasize the role of exceptions and exception handling by using the phrase "signalling an abort" to indicate that a fault has been observed and an appropriate exception raised. We will also use the term "recovery handler" to designate exception handlers responsible for handling aborts.

From the perspective of recovery, actions are required to either execute completely or never begin execution. When an action is declared to exhibit fault recovery, provisions must be recorded on stable storage to bring about a correct outcome should the action fail. Enough information must be recorded on stable storage (accessible to the computation performing the recovery) to ensure either that the effects of the action can be undone or that the action can be forced to a completion state in spite of its failure.

The problem of recovering from failed actions in C^3I systems can be understood by contrasting it with the similar, but simpler, problem of recovering from failed transactions in database transaction processing. In a database transaction processing an action, once begun, may either terminate successfully or abort. If the transaction aborts, recovery is carried out by undoing all of the actions effects. If the action terminates successfully, it is expected to commit. By committing, an action makes its effects permanent and they cannot be undone by rolling them back.

A commit in database transaction processing is an indivisible operation. The act of committing is usually reduced to modifying of a single block on secondary storage. Thus, an action is either uncommitted and none of its effects are permanent, or it is committed and all of its effects are permanent. In the case of database transaction processing, all of

the effects of the action are limited to changes on secondary storage, and it is possible to first make the change, and then decide whether to make it permanent.

In command processing, however, the committing of an action is a more difficult process. An action may commit before it terminates (either by executing an explicit commit operation or by executing an irreversible operation), thus requiring that recovery force the action to completion. Furthermore, the act of committing does not reduce to the modifying of a single byte on secondary storage. Instead, the commit may be a consequence of having caused a state change in a physical system. The action may set a bit before changing the state of the physical system. This bit merely declares the action's intention to commit. The action may set another bit after the state of the physical system has changed. This bit declares that the action has committed. Unlike the case of a commit in database transaction processing, the setting of a bit did not in and of itself make the state permanent. A fundamental problem in generalizing from transaction processing to command processing is the issue of determining whether an action aborted before or after the action caused the state of a physical system to change. If the action aborted before it set the bit declaring its intention to change the state or after it set the bit indicating that it has completed the state change, the answer is obvious. If the action aborted after it set the first bit but before it set the second, additional steps must be taken to determine whether the state change occurred.

Recovery may proceed in one of two ways. The two possibilities are denoted by the terms "backward" and "forward" recovery. Backward recovery returns the computation to an earlier state which existed prior to the failure. Generally this involves undoing all of the aborted action's effects thereby creating the illusion that the action had never run. Forward recovery puts the computation into a new state from which computation may continue. The recovery mechanism may construct this new state by forcing the aborted action to completion or otherwise constructing a state which could have resulted had the computation not been interrupted by the fault. Forward recovery may also construct a new state which incorporates the fact that a fault occurred, e.g., a new state in which the faulty subsystem is shut down and the services it had provided are no longer available.

To facilitate these uses of forward recovery, we allow an action's recovery handler to be selective with respect to which objects and data areas are rolled back or modified during recovery. The recovery handler may be selective in aborting other actions which depend in some way on the failed action.

In C^3I, forward recovery is used in two ways. First, recovery procedures may be used to increase the reliability of a computation in spite of any undetected flaws which may remain in it. In this case the recovery handler is responsible for reparing the fault and allowing the action to continue executing. This is the required approach if an action has been allowed to commit (perhaps by performing an irreversible operation) before it terminated.

Second, recovery procedures may be used to protect one computation from flaws in any other computations or systems (including physical systems and computer hardware) with which it interacts. The faulting action can signal other actions that a fault occurred. The other actions then would have an option of reconfiguring themselves and so as to avoid conditions in which they might also fault.

In a similar fashion, forward recovery can also be used to prevent the abort of an action which has permitted optimistic reads from resulting in a cascade of aborts among processes who have, directly or indirectly, used the uncommitted data.

The recovery procedures must be supported by choices in the design of the hardware, the software, and the data stores. Used in this second way, recovery can provide firewalls to prevent faults from cascading. When faults cascade, each fault triggers other faults in heretofore correctly functioning elements of the system until the system as a whole fails in an uncontrolled and possibly catastrophic way. The next section discusses extensions to this basic model and introduces the syntactic structures we use for expressing fault tolerant designs.

2.2 Extensions to the basic model

In adapting the action/object model to the needs of embedded, real-time systems we have made several modifications to the approach taken in Clouds.

- We regard recovery as the responsibility of the action rather than of the objects touched by the action —thus, we attach recovery handlers to the action rather than abort handlers to the objects.
- We emphasize the importance of forward recovery and have added a mechanism for restarting an aborted action at an intermediate stage. The data used by an action is partitioned into objects and data areas. A aborting action may selectively rollback or modify data areas and objects.

- We have added coroutines to the paradigm of action-based programming and allow optimistic concurrency control.
- We provide two mechanisms for optimistic concurrency control. An action may allow other actions to use its uncommitted results. If the action aborts and the uncommitted results are rolled back, any actions having used those results are aborted. An Action may commit some of its results even before it terminates and this can be regarded as a second approach to providing optimism. An aborting action is not allowed to use the roll back mechanism on data it has previously committed. The aborting action is, however, allowed to modify the commited data and may explicity abort actions which have used the committed results.
- We separate failure and concurrency atomicity. We have defined an action as a unit of computation which is subject to system management. In its simplest form, an action is merely a segment in a thread of execution. An action may, however, be coded to exhibit failure and/or concurrency atomicity.

We use these extensions to discuss and solve several problems related to the construction of embedded systems.

2.2.1 Forward Recovery

This section describes our methodology for forward recovery. The methodology is expressed in terms of *objects*, *actions*, and *abort (exception) handlers*. In the course of this discussion, we introduce the syntax of the design language we use when describing fault tolerant actions. The most important langauge features are introduced here. Additional language features are introduced in Part II.

Object An object is an abstract data structure. An object contains data and operations that may operate on the data. An example of an object is a file. The file's contents represent the data, and the operations are *read, write, open, close, ...* Objects are used in a programming methodology that supports abstraction. "An abstraction isolates use from implementation: an abstraction can be used without knowledge of its implementation and implemented without knowledge of its use [18]."

The syntax of an object is

```
implementation of object <object_name> IS
...data...
...procedures and functions...
end
```

Action An action is a unit of work. An action represents a segment along a thread of execution that may pass through one or more objects. An action may invoke nested actions. Each action may have an associated exception handler. When an exception occurs, the action jumps to the statement *on abort* and executes the exception handler. The *on abort* statement is defined within the scope of the action. The exception handler processes according to the *stages* which are described below.

The syntax of an action is

```
begin action <action_name>
  stage 1: ...
  ...
  stage n: ...
  on abort
  case stage OF
     1:...
     ...
     n:...
end
```

The code, data, and control information associated with an action is encapsulated in an object of type **action**. The entry points, such as **abort** and **restart**, in the **action object** provide the operating system with the hooks required for action management.

Stages Stages are perhaps the most novel aspect of our language. Each action is divided into zero or more stages. A stage is a label attached to a sub-segment of an action. Stages are a series of labels. An action executes in stage i until until the action passes stage label $i+1$ which causes the action to execute in stage $i+1$. On the event of an exception, control jumps to the beginning of the current stage label in the exception handler. The exception handler is marked with the statement *on abort*. The *on abort* statement is defined below.

on abort The *on abort* statement marks the beginning of a recovery handler. The following action template is used for explanatory purposes in the discussion below:

```
begin data area data_area_name
  <variable declarations>
end data area

begin action action_name
   stage 1: f(x)
   stage 2: g(x)
   stage 3: h(x)
   stage 4: a(x)
   stage 5: b(x)
   on abort
   case stage of
      1: forward recovery 1
      2: forward recovery 2
         resume(3)
      3: forward recovery 3
         raise_exception(exception_name3)
      4: backward recovery 4
      5: backward recovery 5
         raise_exception(exception_name5)
end
```

The recovery handler may operate in the ways listed below:

- **Total forward recovery and termination:** When an action raises an exception, the recovery handler processes the exception by invoking the recovery routine marked by the appropriate stage number. The recovery routine may clean up the exception and return with, or without raising its own exception. If the recovery handler does not raise an exception, then we say the action is *totally recovered and terminated.* We use this terminology because the recovery handler believes that the exception can be screened from its invoker, thus protecting against a cascaded exception. The stage 1

recovery handler of the template is an example of this case because an exception is not raised.

- **Forward Recovery and Resumption:** In this case an action raises an exception, and the recovery handler executes some forward recovery. The recovery handler then executes the statement *resume(i)*. The *resume* statement overrides the default of the recovery handler (terminate), and resumes execution at the beginning of stage(n). The stage 2 recovery handler of the template is an example of this case. Here, some forward recovery is executed and control is passed back to the beginning of the action's stage 3.

- **Partial Forward Recovery and Termination:** In this case, the action does as much forward recovery as possible, but is either unable or unwilling to shield its invoker from the exception. Here, the action terminates and signals its invoker that an exception has occurred. Stage 3 of the template is an example of this case.

- **Backward Recovery without exception:** This case models actions that have little to no functional criticality. If the action proceeds correctly, then the action terminates correctly. Otherwise, the action precedes until an exception is raised, and rolls back to its state at the time the action was invoked and returns.

- **Backward Recovery with Exception:** This case is identical to *backward recovery without exception*, with one difference: an exception is raised. This case is used when atomicity semantics of an action is required. Here, an action either precedes to completion, or rolls back to its initial state at the time it was invoked. If the action rolls back to its initial state, then an exception is raised. The exception is handled by its invoker.

2.2.2 Coroutines

We use the cobegin ... coend construct to express coroutines. Brakets are used to delimit the individual coroutines. We use locks, semaphores and rendezvous to express the interaction of coroutines (and more generally of processes).

A skeleton for a coroutine follows:

```
begin action
```

```
stage 1:

stage 2:

stage 3: cobegin

          A: { stage 1:

               stage 2:
             }

          B: { stage 1:

               stage 2:

               stage 3:
             }

stage 4:

on abort



end action
```

To refer to stages within a coroutine it is necessary to refer to the stage containing the coroutine statement and the label of the coroutine. For example, resume(3.A.2, 3.B.1) will resume execution at stage 2 in coroutine A and stage 1 in coroutine B.

2.2.3 Concurrency control and concurrency atomicity

Concurrency atomicity is the property that actions execute in a serializable manner. An action is implemented by a sequence of operations on objects. The global state of each invoked object is visible to currently executing actions. The operations of concurrent actions may be interleaved, but if concurrency atomicity is to be preserved, then the sequence of operations performed by the actions must be serializable. The designer of an object specifies the locking protocols to be used when accessing the object in order to achieve serializability. It is possible to specify the locking in ways which allow nonserializable interactions. With the use of top-level actions it is possible to approximate the effects of optimistic concurrency control, i.e., the results of an action can be made visible before the action commits provided the results to be made visible were the consequence of a nested, top-level actions. Cascading aborts, a drawback to optimistic concurrency control, are avoided by not allowing recovery of objects touched by a nested top-level action after it has committed.

Concurrency Control Whenever two or more processes have the potential to invoke the same object, then synchronization problems may occur. Objects in our language are *passive* in the sense that they have no associated process. For example, in conventional systems, different users may have access to the same text editor. In this case, all of the users share the same image of the text editor's code, but supply their own local space for storage of their edited files. In our language we call the text editor code an object. The text editor code may operate on local variables (files).

We extend the notion of shared code by associating a state with an object. Suppose an object exists that is a text editor for a special file *foo*. Then, the text editor object would be composed of the text editor code and the file *foo* itself. If two or more users each wish to simultaneously edit file *foo*, then the users both invoke the text editor object simultaneously. Obviously, there is a potential for concurrency problems.

We address the concurrency problem by separating the language constructs used for synchronization from the language constructs used for concurrency control. A user may associate synchronization variables with either objects or actions. In our text editor example, if the the text editor object has its own synchronization variables, then actions that invoke the object need not synchronize. Unfortunately, one may find programming such an object difficult if the object must be guaranteed to assure correctness (e.g. serializability

and deadlock avoidance). Therefore, we allow actions as well as objects to use synchronization variables. By placing synchronization variables in actions we avoid the problem of attempting to create fault tolerant *objects*, but we introduce the problem of creating fault tolerant *actions*. We believe fault tolerant actions are easier to build than fault tolerant objects, because actions tend to be more special purpose in nature.

If two actions share the same object and one action aborts, then what should the other action do? Our language is a general purpose language that can express any correctness criterion. In the paragraphs below we present various correctness criteria for recovery and their associated solutions.

Serialized, Low Performance In situations in which all actions must be serializable, but good performance is not highly critical, we omit forward recovery from all actions. When an action aborts, the action rolls back each touched object to the object's original state (the state at the time the action invoked the object). All other actions that touched the objects also abort.

Non-serialized, high performance In some situations, non-serializable execution may be allowed. In our missile example, all that matters is the final position of the missile. If the history of the missile's movements turn out to be non-serializable, then we do not care. In this case, we may use forward recovery whenever deemed appropriate.

Serialized, high performance Designing recovery schemes that achieve both serializability and high performance is an open research problem. A quick survey of some of these schemes are provided below: A transaction T_i *reads from* T_j in H if T_i reads some data item from T_j in H. Notice that it is possible for a transaction to read a data item from itself [4]. We denote a commit point c.

- **RC:** "A history H is called *recoverable (RC)* if, whenever T_i reads from T_j $(i \neq j)$ in H and $c_i \in H$, $c_i < c_j$. Intuitively, a history is recoverable if each transaction commits after the commitment of all transactions (other than itself) from which it read [4]."
- **ACA:** "A history H *avoids cascading aborts (ACA)* if, whenever T_i reads x from T_j $(i \neq j)$, $c_j < r_i[x]$. That is, a transaction may read only those values that are written by committed transactions or by itself [4]."

- **ST:** "A history H is *strict (ST)* if whenever $w_j < o_i[x]$ $(i \neq j)$, either $a_j < o_i[x]$ or $c_j < o_i[x]$ where $o_i[x]$ is $r_i[x]$ or $w_i[x]$. That is, no data item may be read or overwritten until the transactions that previously wrote into it terminates, either by aborting or committing [4]."

Commit before terminate We allow actions to commit (either in entirety or merely selected data areas and objects) before they terminate. An action may commit either explicitly or implicitly. An implicit commit may occur before termination if the action performs an irreversible operation on the physical system being controlled. If an action aborts between the time the action commits and the time the action terminates, the action is required to execute to completion, and it is the responsibility of the recovery handler associated with the action to see that the action does indeed complete. "Completion" can be either restarting the action or merely leaving the objects and data areas touched by the action in a consistent state.

As an example, consider the following high-level control loop of a missile. In our example, there are three stages. The first stage, validates the status variable against a set of internal sensors. If the status variable is incorrect, then the status variable is updated, and control resumes in stage 2. Stage 2 is an example of a stage that has the potential to terminate before it commits. Here, the missile's position is moved. Internal sensors determine when the correct end state is reached. If for some reason, the end state cannot be reached, then the exception *incorrect_position* is raised. The status of the action reflects the final position of the missile. In this case we say the action "commits before termination" because the action does not proceed all the way through stage 3, but the action does change the state of the missile (reflected by the variable *missile_pos*). The final stage is merely a check against possible programmer errors; the final stage *should* be a "no operation."

```
begin implementation of object missile
begin data area mis
   missile_pos : position_type
end data area

procedure missile(IN next_pos : position_type)
begin
```

```
 begin action missile_act
   stage 1: validate(missile_pos) !check status with missile sensors
   stage 2: while not missile_pos = next_pos
               move missile to next_pos from original missile_pos
            end
   stage 3: validate_end(next_pos,missile_pos)

   on abort
   case stage of
      stage 1: initialize
               resume(2)
      stage 2: raise_exception(incorrect_position)
      stage 3: raise_exception(unable_to_figure_out_problem)
   end
 end
end
```

2.3 Failure atomicity

Failure atomicity is the property that an action appears to either execute entirely or to not have executed at all. The question of what it means for an action to execute entirely is a difficult one. We apply an informal notion. An action has executed if it has changed its parent environment. An action has executed completely provided it has changed left the parent environment in a consistent state. A state is not consistent if it will cause subsequent actions to fail. For example, an action is supposed to close any files it has opened but fails to do so (i.e., it has not executed completely): other actions may incorrectly assume the files are closed and, because that assumption is violated, execute incorrectly. If, however, the action had discovered that it was unable to close the files and, instead, set a flag known to the other actions to indicate that fact, then it can be regarded as having executed completely. The question of failure atomicity is closely connected to the question of precedence among actions and among operations. This discussion is taken up in section 2.3.

Failure atomicity is provided in Clouds by means of backward recovery. In backward recovery, the effects of an aborted action are undone and the illusion is created that the

failed action never began to execute. Clouds provides several loopholes in failure atomicity: nested actions can be marked as top-level actions and objects can be equipped with "roll your own" recovery handlers. When an action aborts, the default recovery procedure is to replace the state of each object touched by the action with a copy of the state as it existed before the action began executing. This default recovery procedure can be overridden by attaching an abort hander to an object. If an abort handler is present, then the object will be recovered not by restoring its previous state but by executing the abort handler. The effects of an action which has been marked top-level cannot be backed out even if its parent action aborts, so objects touched by nested, top-level actions will not be restored. A chief advantage of marking nested actions as top-level is that the results of a top-level action are visible when the top-level action commits even though the parent of the top-level action has not yet committed.

Precedence relationships among actions The precedence relation must be qualified whenever it is possible for actions to fail. The possibility that an action may means that either of the actions in the precedence relation may not occur. The assertion that A_1 precedes A_2, unless qualified, means that if A_1 and A_2 both commit then A_1 must precede A_2. Either A_1 or A_2 may commit without the other. This is termed weak precedence. There are two consequences of weak precedence:

1. A_1 must precede A_2 if A_1 is to occur at all, and
2. A_2 must follow A_1 if A_2 is to occur at all

The precedence relations between two actions may be strengthed with either of the following qualifications:

1. A_1 must precede A_2 if A_2 is to occur at all, or
2. A_2 must follow A_1 if A_1 is to occur at all

These two qualifications, when taken together, define the concept known as nested atomic actions. This we call strong precedence.

The weaker forms of precedence arise as possibilities when two processes interact and an action within one process must precede an action in the the other.

The abort handler can ensure that the precedence among operations is preserved in spite of failures.

For example, if one knows that a particular action is *idempotent* (an operation f is idempotent provided $f(x) = f(f(x))$) this fact may be used during recovery. Thus, if the abort handler is unable to determine whether f has been invoked, it can invoke f during recovery.

Suppose the precedence relationships among operations may require that particular action **A** f_1 be followed by f_2. Suppose further that a particular execution of **A** aborts after executing f_1 and after executing an irreversible operation. Finally, suppose that, because of the failure, it is uncertain whether **A** has executed f_2. If f_2 is idempotent, the abort handler can ensure failure atomicity by executing f_2 with out determining whether it had been executed just prior to the abort.

Some difficulties arise when processes interact through shared objects. It is possible through the use of shared objects for two actions within different concurrent scopes to communicate. If optimistic reads are permitted and the locking is not two-phase, then the actions may communicate in an unserializable way. In such a situation failure atomicity may be preserved but concurrency atomicity is lost.

We term an action with nonserializable iterations with other actions and which cannot be rolled back a *weakened* action. A weakened action is little more that a coroutine with a high powered exception handler.

Weakened actions may contain nested actions which are not themselves weakened. When two weakened actions interact, full strength subactions nested within them should interact in serializable ways. It may be desirable to further constrain allowable schedules to a subset of the serial schedules. Such constraints can generally be expressed using sequencing and rendezvous mechanisms.

A special case: nested actions Nexted actions are are a common extension to the object/action model.

Nested actions are actions at different levels of abstraction. A parent action may spawn a set of subactions. Each subaction appears to the parent action as an atomic unit even though the subaction itself is composed of its own operations and sub-subactions. After

each of the child subactions complete, the parent action may complete. Each child subaction may be the parent for one or more of its own child actions. Normally, the correctness criteria for system state is that the system state must result from a series of serializable transactions. Nested actions provide a weaker correctness criteria that is strong enough for most applications. The correctness criteria for nested actions is that highest-level actions (HLA) must be serializable. Since a user views the system state using HLAs, the user's view of the system state is guaranteed to be consistent because the HLAs are serializable.

Example An example of a history of nested actions that satisfies the weak correctness criteria, but not the strong correctness criteria is provided below: Suppose an HLA spawns three subactions s_1,s_2, and s_3. All of these subactions operate on a list of integers. Suppose the HLA knows that every integer in the list is positive, and subactions s_1, and s_2, are inserting negative numbers into the list. Suppose s_3 computes the maximum of all of the integers in the list. Since s_1, and s_2 add negative numbers to the list, the HLA knows that s_1 and s_2 will not affect the execution of s_3. Therefore, s_1 and s_2 need not be serializable with respect to s_3 in order for the HLA to be assured the "correct" result from its subactions.

Correctness Criteria Griffeth [10] describes the correctness criteria in database systems: "The reason that the executions are correct, in spite of violating this rule [serializability], is that the data-base state with which we are really concerned is just the set of tuples in the tuple file and the set of index entries in the index. Any pair of states having the same sets in the two files must be regarded as equivalent. Therefore, the resulting "top-level" database state is the same in the original executions as in some serial execution of the committed transactions."

Performance The primary motivation for nested actions is they may be used to yield better performance than comparable systems that do not use nested actions.

> A history in a multi-level system is a collection of histories, one for each levels of abstraction. The concrete actions at one level of abstraction are abstract actions at the next lower level (this is just another way of saying that the lower level implements the higher level actions). A multi-level history is serializable and atomic by layers if each layer is abstractly serializable and atomic and if

the committed abstract actions at one level are exactly the same as the concrete actions at the next level.

The key to achieving performance improvements is recognizing the multiple level of abstractions in a system. As the example provided above illustrates, if a problem can be decomposed into different levels of abstraction, then the knowledge of semantics can be used to enhance performance.

Implementation Nested actions are implemented by (1) providing a locking structure as described by Moss [19], and (2) providing a facility to *UNDO* aborted actions. The locking structure assures serialization at a child's level of abstraction is inherited by the parent action. The *UNDO* mechanism is implemented to allow some actions commit while other actions abort. Implementing an *UNDO* mechanism in a nested environment is more problematic than implementing the *UNDO* mechanism in a non-nested environment because the *UNDO* mechanism must operate at different levels of abstraction. If nested actions are "called by higher level actions which are subsequently aborted, then the nested actions which committed must be UNDOne rather than aborted. The lowest-level concrete actions are considered to be committed in this context as soon as they are done. This is consistent with normal usage, because page reads and writes (the usual bottom level actions) are UNDOne, not aborted."

2.4 Some background on failure and recovery

The discussion of fault tolerance introduced the concept of an action that commits before it terminates to represent a permanent state modification that occurs before an atomic action terminates.

Inevitably, problems will develop with both the hardware and software components in a system. The problems with the hardware may result from ordinary wear and tear, from misuse, or from a mismatch between hardware capabilities and requirements. While software does not wear out, problems may develop because of errors in the requirements analysis, systems architecture, design and coding of components, and data used.

These errors will, from time to time, manifest themselves in the operation of the system, and a fault is said to have occurred. When the chain of events stemming from the fault

effects systems functionality or performance as perceived by human users or client systems, a failure is said to have occurred. Specifically, a failure is the cessation of the normal, timely operation of the system or the delivery of incorrect information.

The distinction between a fault and a failure is not sharp. A *fault* can be regarded as the improper execution of a component. A *failure* is a fault which manifests itself to a human user. In this sense, a failure cannot be recovered. It is also possible for one to use the term *failure* in a second sense by speaking of the failure of a subsystem: a failed subsystem is one which has failed from the point of view of its clients, i.e., the other subsystems with which it interacts. The subsystem failure is a fault in that it may or may not manifest itself in the behavior of the system of which the faulty subsystem is a part. We will generally use the term *failure* in this second sense and will speak of a fault as causing an action to abort and invoke a recovery handler. When we wish to refer to the action which has aborted but not yet invoked a recovery handler, we will refer to it as a *failed* action.

Often considerable time may elapse between the internal events we term faults and the anomalous external behavior we term failure. The failure may be the cumulative effect of a number of faults. If software is designed to be fault tolerant, then some portion of the software is given over to preventing faults from causing the system as a whole to fail. It will generally not be possible to prevent a failure, but with proper design it may be possible to delay it, minimize its extent or consequences or at least maintain control of critical subsystems during the failure.

The distinction between detected and undetected faults is an important one and the research literature has addressed both cases. The simplest type of fault is the crash; a crash always results in a failure. A crash is characterized by the complete cessation of activity within a system (or subsystem). A crash is generally accompanied by a loss of any data in volatile storage.[2]

Because of this problem, a fault tolerant computation is generally required to periodically flush data to stable storage. In the event of failure, the computation can be resumed using an intermediate state constructed from the data saved on stable storage. The data on stable storage can be in the form of a checkpoint (shadow) or a log. The computation can be resumed either by restarting the failed subsystem or by transferring critical functions to other subsystems.

[2]Lin has recently shown that data in volatile storage survives a large portion of system crashes [17].

The same strategies can be used to protect against faults which result in anomalous behavior but not in a crash. Once a fault has been detected, it is possible to treat the faulty subsystem as though it had stopped processing altogether and either restart it from a state which was know to be correct or transfer its assignments to other correctly functioning subsystems. The problem of responding to and recovering from faults once they have been detected is known in the research literature as the "fail stop" problem.

Attention has also been directed towards the problem of detecting failure in a subsystem which is still active. A failure which does not make itself known immediately but which must instead be "discovered" is known as a Byzantine failure [16]. With Byzantine failure, a faulty subsystem may intentionally or unintentionally disguise the fact that it has failed and interfere with the normal activity of other subsystems, especially with respect to the task of detecting that the failure has occurred.

While fail stop is studied is a generalization of detectable failure, Byzantine failures are studied as generalizations of failures which have not yet been detected and may prove difficult or elusive.

Our interest is neither in fail stop nor Byzantine failure, but rather in protecting system from residual errors in specification, design, and implementation, effects of ordinary wear and tear on the physical components of a system, and the effects of spontaneous external noise. The techniques discussed here should be used not with the expectation that they will prevent failures but with the hope that the mean time between failures can be extended by equipping key components with appropriate safeguards for tolerating certain types of faults.

A fault tolerant system is designed by first choosing the correct hardware architecture to satisfy the most important or most stringent requirements, and then building a fault tolerant system on the hardware base. When building fault tolerant software systems, the first line of defense is to avoid or eliminate as many errors as possible. Unfortunately, it is not practical to remove all errors. Studies suggest, for example, the number of errors in a large system remains constant in spite of ongoing efforts at corrective maintenance. A second line of defense then is to design the code so that erroneous code, state, and data are recognized before they are used in a way which could result in a fault. This is also not always possible, so a third line of defense is required: it is often necessary to recognize that a fault has occurred and cope with it. Ideally, a fault can be handled in a way which does

not impact the performance and function of the system. Sometimes this is not possible and the objective of fault tolerance strategies should be to minimize the impact on performance and function.

2.5 Recoverable actions and objects

In the Clouds model, recovery handlers are bound to objects rather than to actions. Thus, when an action aborts the objects touched by the action are recovered by each object individually. The recovery of objects may, depending on the semantics of the object, involve rollback or the invocation of the objects recovery handler. This is a satisfactory solution when the objective of recovery is to construct a state in which it appears that the aborting action had never been invoked. In order to insure availability of system resources, many system objects are unlocked before an action which has modified them commits. In such cases concurrency atomicity may be preserved but failure atomicity is lost.

The onus is on an object's programmer to provide a "roll your own" recovery handler when an object may represent a loophole in failure atomicity. The recovery handler must allow the effects of the aborting action to be backed out without interfering with subsequent operations on the object by other actions. For example, if an object represents a resource pool, then the recovery handler must return any resources checked out to the aborting action. This can be done without interfering with other resources which have been assigned to other actions.[3]

For objects with richer semantics this solution is not appropriate. Suppose the shared object is an index into a file. Changes to the index by one action may directly affect the execution of other actions. If the first action aborts, there are two problems: how to back out the changes to the index and how to notify the other actions that their views of the index are invalid. We believe the appropriate solution in such a situation is a recovery handler which is sensitive to the semantics of the aborting action.

In our model, recovery handlers are bound to the actions. Thus, when an action aborts,

[3]The recovery handlers described in this handbook can be used to implement Clouds recovery handlers. A Clouds recovery handler is associated with an object and is not sensitive to the semantics of the process containing the aborted action. A Clouds recovery handler is implemented using a special entry point designated as the *Recovery Handler*. Now, if an action aborts after having touched an object, the recovery handler in the action can invoke the *Recovery Handler* entry point in the object.

the action's recovery handler is free to decide which objects are to be recovered and how that recovery is to be achieved. This allows the recovery process to be sensitive to the semantics of the aborting action. In particular, it allows recovery to force to completion an action which implicitly committed before aborting (terminating). It also allows actions to restart at intermediate stages.

In the case of the action which aborted after modifying a shared index, several options are available. For example, the index could be rolled back completely and all actions sharing the index aborted and restarted. A second option would involve regarding the aborting action as already committed and forcing it to completion. In this case the recovery handler must initiate compensatory operations. These compensatory actions may include corrections to the index and signals to other processes which may have used the incorrect version of the index. The actions which used the incorrect index may themselves proceed in several ways. An action may be robust and have recognized and handled the errors in the index on its own. It may abort when it receives a signal that it was using an incorrect version of the index. It may either rollback and restart itself from the beginning or may restart in an intermediate state after identifying the inconsistencies between its view of the index and the current version of the index.

Of course when the original action recovers, it need not choose to recover the index at all; instead, it may attempt to restart in an intermediate state which assumes the index has already been modified. If the original action aborted because it was involved in a deadlock, because recovery was initiated in some other object which the action had accessed, or because of a hardware failure, this would be an appropriate strategy. A similar strategy would be possible even if recovery involved swapping in a new version of the code for the aborting action. This important ability of an action to restart in an intermediate stage provides a firewall against cascading aborts.

As a third possibility, the aborting action could pass the burden of recovery and/or continuation to its parent action.

In our model the onus is on the programmer of the action: he must have a correct understanding of how his action interacts with the persistent objects in the environment and how his action interacts, albeit indirectly, with other actions sharing these objects. We suggest that the designer of an object include sample recovery protocols as a way of coordinating recovery among actions sharing an object.

3 Requirements for Fault Tolerance

A system's capabilities are defined in terms of requirements. Some of these requirements involve the system's ability to recover from faults. We call these requirements *fault tolerance requirements.* In this section we identify many system requirements that are also fault tolerance requirements.

The purpose of this section is to establish a connection between fault tolerance requirements and software design. This section presents a taxonomy of fault tolerance requirements. For each requirement we present forward references to sections within this handbook that provide detailed descriptions of mechanisms that may be used to satisfy a given fault tolerance requirement. The forward references serve as a framework for choosing good fault tolerance mechanisms.

A system's requirements define constraints on a system's operations. Requirements tend to be informal and do not specify a particular system. The task of constructing a functional description that satisfies a given set of requirements is relatively complex. The functional description must satisfy the following constraints:

1. The functional description must satisfy the requirements.
2. The functional description must be implementable.
3. The functional description must have enough detail to establish a formal system design specification.

In order to create a functional description, one must translate abstract requirements into concrete functions. The concrete functions must be expressed in a notation that allows formal designs to be constructed. For example, a real-time requirement for a data collection facility may state that the data collector must operate under a real-time constraint of 5 milli-seconds. A functional description of a particular system must translate this abstract requirement into a functional description of a protocol that is guaranteed to satisfy the real-time requirement.

Translating a set of requirements into a functional description is non-trivial if the requirements stipulate some degree of fault tolerance. For example, suppose a requirement mandates that a data collector will be operational 99% of the time. The functional description must take to following questions into account:

- Which data collection operations must be guaranteed to run to completion?
- Which facilities must be operational 99% of the time, and which may fail?
- Which facilities must communicate, and what happens if one fails?
- How fast must the distinct facilities operate in order to be considered "operational"?
- What kind of off-the-shelf equipment is available that satisfies the requirements?

In the sections below we generalize these questions to form a characterization of the system requirements that are also fault tolerance requirements. These requirements lie in one or more of the following categories:

- **Functionality:** functional requirements form the statement of the results the software system is to compute in response to its inputs or the commands it is to issue in response to conditions in its environment.
- **Atomicity:** requirements related to atomicity stipulate the sets of commands (actions) which must be executed as a group (i.e., failure atomicity) and the extent to which groups of commands may interact (interfere) with each other (concurrency atomicity)
- **Synchronization:** requirements related to synchronization stipulate the extent to which precedence relations among actions must be maintained and the degree of concurrency which must be allowed.
- **Time:** requirements related to time are timing constraints. Timing constraints specify the maximum and minimum amount of time which can elapse between two related operations.
- **Processing:** processing requirements stipulate performance constraints (e.g., time allowed for a procedure call, time allowed to access a record in a file, the rate at which database transactions must be processed). From these processing considerations requirements can be developed for the hardware and software environments within which the software is to operate.

Time and functionality are the most important requirements. Atomicity and synchronization are important considerations when deciding how bet to satisfy the reqirements related

to time and function. Processing power has an effect on the ability to satisfy the other requirements.

There are natural tradeoffs among these requirements. For example, if it is critical that a particular timing constraint be satisfied, then it may be necessary to sacrifice concurrency atomicity. To satisfy the constraint, it may be necessary to allow a high priority action to preempt the low priority action with out allowing it to recover. The consequence may be to compromise the ability to perform a minor function until the relevant portion of the system is reinitialized.

From the system-level requirements it is possible to infer requirements for the mechanisms which provide fault tolerance. For example, if it is more important that a parameter be adjusted with a particular frequency than that adjustments be of a particular precision, then this has implications for how fault tolerance should be provided. In this case the process making the adjustments should be highly available and resilient even if its reliability (measured by the accuracy of the adjustments) is compromised. The following criteria can be used to describe the requirements for fault tolerance:

availability: physical computational resources and data are made available in spite of failure in servers which provide access to them or support their use.

resiliency: faults do not cause a computation to terminate in an abort

reliability: faults do not cause a computation to produce an incorrect answer

forward progress: recovery does not always undo all the work which had been completed before the fault.

The criticality of each metric taken individually affects the design of fault tolerance mechanisms in terms of the choice of hardware, the choice of software model, and specific decisions regarding software architecture.

Once the requirements for the mechanisms for providing fault tolerance have been identified, the mechanisms themselves must be designed. The design choices include decision about what elements of the system (hardware, software, and data) should be supplied redundantly and how those redundant elements should be exploited during recovery. Sections four and Part II detail a number of design options. The remainder of this section provides

additional detail on the interaction of system-level requirements and requirements for fault tolerance.

3.1 Functional Characterization

The functional characterization specifies the type of functions dictated by system's requirements. The functional characterization is expressed in terms of three parameters: time preservation, frequency of updates, and criticality.

Time Preservation *Time preservation* dictates how long collected data must be preserved. Fox example, a real-time data collection device typically does not store collected data. Instead, data is immediately transmitted to a long-range storage device for processing. The long-range storage device may store the collected data in main memory, a disk, or backup tape.

A variety of software structures are used to preserve data: local variables, parameters, global variables, objects, and system structures. These structures are tools used to preserve data for varying lengths of time. This paper focuses on *objects* (see section 2.2.1) because we feel that the concept of an object is a general purpose notation that can describe any kind of data no matter how long it must be preserved.

Frequency of Updates *Frequency of Updates* states how often data is modified. An example function that updates data every micro-second is a fast real-time data collector; an example of a function that never updates data is a periodic backup onto magnetic tape (the periodic backup never overwrites a tape).

Criticality This parameter is a qualitative measure of the criticality of a function. The possible values range from *not critical* to *maximum criticality*. Functions that are deemed not critical should not have a large impact on the fault tolerance functional description. Functions that are highly critical have maximal impact.

Functions that are highly critical should have an associated recovery mechanism. The recovery mechanism discussed in this paper is an exception handler attached to an action. We describe this mechanism in detail in section 2.2. An example of a highly critical function

is a life-support system. An example of a low criticality function is the UNIX[4] **rwho** command that returns the login ids of currently logged in users on a set of interconnected machines. The **rwho** command may omit users who have just logged in recently and should return a result even if current information is not available for some machines.

3.2 Atomicity

Current trends in system design indicate that separating a system into modules is essential to creating reliable systems. Each module is expected to be self-contained, and the module's functional specification defines its output given each input. The hierarchical design methodology recommends that modules be recursively defined in terms of sub-modules. A given requirement may mandate that a particular module operate as an atomic unit: an operation implemented by the module either runs to completion, or does not run at all. We call an operation implemented by an atomic unit an *atomic action* (see section 2.2).

A functional description defines the operations of each atomic function. The formal design document describes the mechanisms that implement the atomic function. These mechanisms may be implemented in either hardware or software. Hardware mechanisms include switchover to backup processors and battery backed-up memory. Software mechanisms include object-oriented programming techniques and exception handlers. Section 2.2 is a detailed description of a methodology that integrates these action-based software mechanisms.

We characterize atomicity in terms of two sub-parameters: size of atomicity and criticality of atomicity.

Size of Atomicity *Size of atomicity* specifies the magnitude of the atomic operation. Values range from a single machine instruction[5] to an arbitrarily large number of operations seperated by an arbitrary length of time. The *size of atomicity* parameter defines the size of the unit that must be recovered. The amount of time required to implement an atomic operation is usually proportional to the size of the operation. An example of an action with a large size of atomicity is a complex database query. The commands to open and close a file

[4] UNIX is a Trademark of Bell Laboratories.

[5] More precisely, the set of micro-operations required to execute the instruction must exhibit failure atomicity.

may be regarded as belonging to the same atomic action (for purposes of failure atomicity). An example of an action with a small size of atomicity is single machine instruction.

The recovery technique may either be *forward* or *backward.* These techniques are described in detail in sections 2.2 and 2.3, respectively.

Criticality of Atomicity Criticality of atomicity is a prioritized range extending from not critical to maximum criticality. Criticality of atomicity should not be confused with functional criticality. A requirement for low criticality of atomicity, but high criticality of function states that an action should be invoked even though there is a risk that the action will leave the global environment in an inconsistent state (e.g., some of the data structures indicate a file is closed while others indicate it is open). Such a situation might arise if the aborted action had committed before it terminated and then recovered incorrectly. In section 2.2.1 we describe a strategy for recovering from actions that commit and then abort. The criticality of atomicity parameter does not dictate a mechanisms for fault tolerance, rather, this parameter is used as part of a cost-benefit analysis to determine whether or not fault tolerance is required for a specific function.

3.3 Synchronization

Synchronization defines how distinct processes communicate. For example, a distributed database may use voting to synchronize updates, while a multi-location data collection activity may use fast, special purpose (hardware-defined) communication. Synchronization is defined in terms of two parameters: criticality of synchronization, and number of distinct functions. This paper does not discuss synchronization specificly, although synchronization is one of the motivations for nested actions (see section 2.3).

Criticality of Synchronization This parameter qualitatively describes the importance of synchronization to correct functionality (note the parameter does not express the importance of correct functionality). Values for criticality range from *not critical* to *maximal criticality.* An example of a function that requires low criticality of synchronization is the previously discussed UNIX *rwho* function. *Rwho* sends a message to each UNIX machine in a local area network requesting a list of users who are currently logged in. If synchronization is not correct, then *rwho* may leave out a few names or return out of date data, but

this is not considered important. An example of a function that requires high criticality of synchronization is a distributed database. Since data integrity is important for correct functionality, correct synchronization must be maintained to assure that data updates are implemented correctly.

Number of Distinct Functions This is a quantitative parameter that describes the number of functions (processes) in a synchronization group. For example, in the distributed database described above, the number of single- processor databases is the number of distinct functions. This parameter is modeled in software using the technique of object replication (see section

3.4 Time

Time requirements is a quantitative measure that describes the real-time constraints of a function. This parameter affects the choice of processor, redundancy approach, and recovery technique. For example, if the real- time requirements are relatively fast, then a relatively fast processor must be chosen. Fine-grained nested actions (described in section 2.3) may be used to speed-up recovery in order to satisfy the time requirements.

The time requirements are expressed in terms of two parameters: real-time, and maximum response time for recovery.

Real-Time This parameter denotes the real-time requirements of a function. A function operates correctly only if the function's response time is faster than the *real-time* parameter. For example, a data collection device attached to a real-time sensor operates correctly only if data is collected before the data is over-written by the sensor.

Maximum Response Time for Recovery This parameter quantitatively describes the maximum amount of time that may elapse before recovery from a fault completes (see section 2.2.1).

3.5 Processing Requirements

The processing requirements define the type of processing required by a particular function. This category is expressed in terms of two parameters: operating system, and memory. We do not discuss this parameter in detail in this handbook.

Operating System This parameter describes the type of constraints imposed upon the operating system. Possible values for this parameter are: real- time, secure, and fault tolerant.

Memory This parameter quantitatively describes the memory requirements.

4 Fault tolerant designs for embedded systems

Embedded systems interact with physical systems within real-time constraints. An embedded system accepts data from sensors and, after some computation, generates control signals. The data from a sensor may by itself be ambiguous, and must be interpreted in the context of other elements of the state of the system and perhaps even in context of recent events within the system. The embedded system will maintain a model of the state of the system being controlled. This state information will describe not only the system's current configuration but other information about its current capabilities, recent events within the system and recent commands issued to it. The embedded system is also likely to maintain a model of the environment in which the control system is operating.

4.1 A model architecture for embedded systems

An event loop is generally the top-level control structure in an embedded system. The event loop monitors the status of various devices and of the physical system being controlled. When a device or an element of the physical system changes state, an event is said to have occurred. The event loop invokes the appropriate event handler.

The event loop may be used in conjunction with a system of interrupt handlers. The interrupt handlers will generally be reserved for handling important asynchronous events. Many of the more ordinary events cannot be detected except in context and require the event loop to compare sensor data against a world model.

The event handlers may or may not make use of a feedback loop. When a feedback loop is used, deviations from expected behavior may be regarded as an event and require the event handler to be reinvoked. This strategy, however, requires a means for dynamically defining events to be monitored for by the event loop. A second strategy is for the event handler to spawn processes in response to events and then these processes are responsible for managing the feedback loop and executing the planned response to the event.

In many embedded systems, it is important for the software to maintain data structures which model the state of the physical system and the state of its environment. The event loop is then responsible for maintaining these models in response to external events. When an event requires a response, the event loop invokes the appropriate action or spawns the appropriate process. The actions and processes which are responding to events must share

access to the models of the physical system and its environment.

4.2 Irreversible operations as a design problem in embedded systems

In embedded systems some operations may be irreversible, i.e., an operation's effects may not hidden until the action containing the operation terminates and commits. In effect, an irreversible operation causes an action to commit before it terminates.

Irreversible operations may occur in three ways:

1. operations may be performed on nonrecoverable objects;
2. operations may trigger events in the physical systems being controlled; and
3. effects of operations may be made visible to other actions before the containing action commits.

This last type of irreversible operation occurs when actions used to define allowable concurrency are larger-grained than those used to define fault tolerance, i.e., when optimistic concurrency control is used.

An irreversible operation is one which cannot be "undone;" that is, it cannot be undone by rolling back the objects and data areas touched by the containing action;

We can distinguish several degrees of reversibility

1. an inverse operation exists and recovery involves executing this inverse;
2. the operation is compensable and recovery involves executing the compensating procedure;
3. an invariant exists and recovery involves performing operations which establish consistency over several variables or objects (this suggests that recovery may proceed even when the source of the fault is unknown); and
4. operations for which recovery is not possible.

An operation may become more irreversible as its consequences are realized. For example, the effects of raising the temperature in some chemical process may be reversed by lowering

the temperature provided the temperature is reduced within a few seconds of being raised. If more time elapses, however, the calories needed to initiate the reaction will have been transferred, at which point the operation of raising the temperature no longer has an inverse. Even then it may be possible to compensate for the change in temperature by other means removing the catalyst, adding other materials, or draining the vat). As the reaction progress, even such compensating actions may not be effective and recovery is no longer possible.

A general programming strategy will be to defer invoking an irreversible operation until all the possible software failure points have been passed. It is also desirable to put guards on the irreversible operation; these guards will check that the conditions for invoking the operation are indeed satisfied. We do not want to invoke an irreversible operation just because an ordinary software error led program execution down the wrong path.

These precautions, however, may not be feasible. Even if they are, the may not be sufficient. the precautions may be inadequate because the fault was:

- in the computation leading to the invocation of the operation but the information needed to verify that computation was not available until after the irreversible operation was completed. Perhaps the needed information was among that returned by the irreversible operation itself;
- the result of a sensor error. the sensor error may not be detected until after the irreversible operation is completed;
- in the irreversible operation itself. the physical system being controlled may not respond to a command as expected. This may be becauses of influences on the physical system other than the control software; or
- in an operation which uses the results of the irreversible operation but faults for other, unrelated reasons.

4.3 Sources and types of faults

The choice of recovery strategy depends, in large measure, on the source and type of the fault. Below is a list of several possible errors which may lead to faults as well as several ways in which faults may manifest themselves.

Most system-level exceptions belong to one of the following categories:

- *Logical Errors:* The program can no longer continue with its normal execution due to internal conditions such as bad input, data not found, overflow, or resource limit exceeded.
- *System Errors:* The system has entered an undesirable state (e.g., deadlock), as a result of which the program cannot continue with its normal execution. The program, however, can be reexecuted at a later time.
- *System Crash:* The hardware malfunctions, causing the loss of the content of volatile storage. The content on nonvolatile storage remains intact.
- *Disk Failure:* A disk block loses its content as a result of either head crash or failure during the data transfer operation [15].

Faults may be transient or nontransient. They may also be the cumulative effect of a number of small errors or the result of a single design error. When handling a fault arising from a transient error it is often appropriate to perform recovery and resume computation as though nothing had happened. If the fault resulted from the cumulative effect of a number of small errors, it may be appropriate to reinitialize some objects or subsystems before resuming computation.

Hardware errors can be handled in three ways depending on the criticality of the functions effected. When handling hardware errors, there are three appropriate recovery strategies: wait for the failed hardware to be returned to service, move the computation to another piece of hardware, or, if the fault resulted from a transient error, restart the computation with minimal reconfiguration of hardware. These alternatives are appropriate whether the hardware device is a memory chip, a printer, a disk drive, or a site in a distributed system.

In all cases it may be necessary to make some associated changes in the software being run or to repair data that was damaged when the hardware failed. The hardware failure may make some information temporarily unavailable and may have caused the permanent loss of yet other information. Much research in distributed systems has been devoted to the problem of moving computation to other hardware in the event of a hardware failures and site crashes. Some of this reasearch has studied the use of replicated data to circumvent the problem of temporarily unavailable data. This research has also addressed the problem of reintegrating the replicas once the hardware problem is resolved.

Logical errors are of two types, software and data. Data errors are tricky to handle in that they may be a consequence of software errors. The data error can be repaired and

computation resumed. Furthermore, effort be made to track down the software element which caused the data error; perhaps with intent of bypassing it until it is repaired.

When dealing with data errors it will generally not be clear why the data has become corrupted. It may have been corrupted though the actions of an outside agent or, in the case of shared data, as the result of an unknown software error. When dealing with a data error, three strategies are possible: 1)determine the correct values and restore them, 2) establish a consistent set of values (this is a weaker condition than correctness), and 3) mark the corrupted data as unavailable. When data is changed or marked as unavailable, this must affect the execution of actions needing the data: in our view they should abort and find an alternative path during recovery. When it is recognized that data is corrupted, it is necessary to notify processes which have used that data. Each process may then decide how to proceed.

Several strategies are available for handling software faults: 1) rerun the computation with the expectation that the fault was transitory, 2) reinitialize the unit and rerun the computation with the expectation that the fault was cumulative, 3)allow the faulty software to continue to operate but patch the results before they are made available, 4) shut down the faulty software and use an alternative unit with equivalent functionality, 5) shut down the faulty software and use an alternative unit with reduced functionality, and 6) shut down the faulty software unit and allow the system to operate in a degraded mode. Of course, the effect of these strategies is to bypass errors or compensate for fault. This should be regarded as temporary measure in effect until the error is repaired.

An error in the interface between control software and the system being controlled may be of several types:

- **tolerance:** the software cannot adjust to the system finely enough or cannot distinguish between two system states;
- **timing:** the responses to changes in the physical system are not appropriate at the time at which they are applied (the response may be too soon or too late);
- **limits:** unexpected response or combination of responses from the physical system may drive the software beyond its limits— buffers may overflow, or there may not be sufficient resources, or there may be a range error.

Such errors may arise because of errors in the requirements analysis. In particular the analytical models used to describe the systems with which the software interacts may have been inadequate. If requirements have been carefully analyzed, the residual specification errors are likely to be subtle, transient ones against which fault tolerant strategies must defend.

Faults in resource management also may arise because of errors in the design of the software. typical faults may include: the unavailability of the required resources, the use fo the wrong resource, contention for a resource, a race condition in getting a resource, a resource not returned to the right pool, improper recombination of fractionated resources, resources not returned, deadlock, resource use forbidden to the caller, resource linked to the wrong kind of queue. One of the purposes of systems layer is to provide these services in a consistent manner to the applications layer.

Faults in resource management can often be handled by reinitializing the resource pool. For this strategy to succeed, the software must have a layered structure. The resource manager can send an abort signal to its clients and thereby reclaim its resources. The aborted clients would, during recovery, rerequest resources. If the clients were appropriately designed, they could resume computation in intermediate states. Thus, the resource manager could reinitialize itself without causing its clients to loose more than a small amount of work.

Faults may also arise because of errors in software architecture, especially errors that are load dependent. Possible errors include the assumption that interrupts will not occur, the failure to mask or unmask interrupts, the assumption that code is reentrant or not reentrant, the bypassing of data interlocks, the failure to lock or unlock data objects, an assumption about the location of a calling or called routine, the assumption that data storage was or was not initialized, the assumption that a variable did or did not change value, inconsistent conventions about the layout and management of data or about the propagation of control information.

If architectural errors are load dependent, then they can often be handled as transient errors. Recovery could also manage the load by shifting computations to other machines or by deferring less critical work.

Faults may also arise because of errors in a software system's internal interfaces: there may be protocol design errors, format errors, inadequate protection against corrupted data,

parameter layout errors, inconsistency conventions as to the meaning of input or return values.

In some circumstances recovery procedures may be able to correct the corrupted data, or resolve the inconsistency in the internal interface. For example, if for historical reasons a system is built using two different parameter passing conventions and there is no discernible pattern governing their use, it would be wise to anticipate the fault by equipping actions with a means of handling the fault, i.e., the recovery handler could attempt to resolve the fault by reformating the parameters.

Faults may also be caused by errors in coding or in low level logic: a wrong operation may be used or missing altogether; operations may be in the wrong order; cases presumed impossible may, in fact, be possible; loops may terminate an iteration too early or too late; cases presumed mutually exclusive may not be, special cases may have gone unrecognized; execution paths may be missing or unreachable; and loops or conditionals may be nested improperly.

Errors such as these will generally require that the computation be shut down unless an alternative, functionally equivalent section of code is available.

4.4 Some basic services

The operating system and run-time system must provide some basic services to support our modifications to the object/action model.

- **Actions:** Actions will interact in serializable ways. Actions may commit before they terminate but may not terminate until all actions which have provided them with data have terminated.
- **Stages:** While an action provides concurrency atomicity, stages provide failure atomicity. Stages have control over what information is logged or checkpointed. If an action aborts, the stage in which it was executing at the time of the abort is recorded.
- **Recovery handlers:** If an action aborts, control will transfer to its associated recovery handler. The recovery handler will be sensitive to the stage at which the action aborted and the type of exception which caused the action to abort. Recovery is transferred to the recovery handler associated with the action containing the stage.

The recovery handler may complete the work of the aborted stage, undo the work of the aborted stage, do or undo the work of other stages and then restart or terminate the action.

The recovery handler may consist of two parts: on abort (triggered when the action aborts) and on restart (executed after the action is restarted but before control passes into the main line of the action)

- **Checkpointing** Checkpointing can be used by an action to save a data area or object state for reference during recovery. An action can explicitly checkpoint at the beginning of a stage and may release a checkpoint at the end of a stage. An action may not proceed with a stage until any checkpoints requested by the stage are written to stable storage or replicated at a remote site as required.

- **Logging:** Logging may also be used by an action to save a information for reference during recovery. A log is an object to which an action can write and from which a recovery handler can read. An action may not proceed with a stage until log entries made by the previous stage are written to stable storage or replicated at a remote site as required.

- **Commit:** "Commit" is used by an action to make data visible to other actions before the first action terminates. A commit represents a promise by the first action not to modify the committed data except in the extreme case that it aborts and is forced to recover.

- **Data invalidation:** During recovery, an action may modify data areas touched during normal execution of that action. If the data area was committed, such modification will invalidate the old values and force and actions which accessed the old values to terminate. If the actions have been properly designed, then cascading aborts should not occur.

- **Pauses:** Pauses may be used by an action to ensure that the data it is reading has been provided by a committed and terminated action. If data is supplied by a committed but not terminated action, then it may later be invalidated. A "pause" causes the consumer to wait until that risk has passed.

- **Windows:** Windows are used in Clouds to map portions of the address space of one object onto the address space of another. We find it useful to allow recovery handlers

to remap windows. This provides a simple way of activating a redundant, backup section of code.

Part II

Constructing fault tolerant actions

5 Introduction to the examples

After an action aborts occurs, the recovery handler must create a consistent state. The state includes data, logical flags and a point at which control is to resume. This consistent state may be one which arose at some juncture before the aborted action began to execute, one which could have arisen had the action not aborted, or one which allows the computation to continue, though in a way which does not hide the fact the action aborted.

The most familiar pattern within this framework is that of restarting the aborted action immediately and in the state which existed just before the aborted action was first invoked. This is the usual practice if an action is aborted because it is involved in deadlock or has been preempted by a higher priority action. If the error was in the code used for the action, then the state is, in database recovery, rolled back to what it was immediately before the action began, but the action is not restarted. It is also common in database recovery to delay the restart of the action until after the reason for the fault has been repaired. For example, if an action needs a resource on a crashed machine, it may either wait for the resource to become available or abort. If it aborts, then it should be restarted once the resource becomes available.

In command processing, however, additional flexibility is required. The options of either restarting immediately, after repairs have been made, or not at all are insufficient. to satisfy timing requirements for critical functions, it must also be possible for recovery to return the system to partial operation immediately and then to full operation once repairs have been made. Thus, the system designer must understand the tradeoffs among system-level requirements for functionality, timeliness, and processing capability. Furthermore, the system designer must understand how requirements for atomicity and synchronization can be modified to balance those tradeoffs. By understanding these tradeoffs, the system designer is able to determine whether particular functions must be resumed immediately following a fault or may wait for repairs to be made. Depending on how a particular function is classified, different approaches to providing fault tolerance are to be pursued.

For functions in which correctness and timeliness are both critical, it must be possible to resume computation in spite of hardware crashes, data errors and faults in related components. The efforts to be expended on providing and exploiting redundant data and redundant software for related components are determined by the relative importance of correctness. If the timeliness with which a function is carried out is critical but accuracy is not, then the emphasis should be on ensuring the computation is resilient and the availability of an appropriate subset of the the resources required. In these cases recovery strategies should emphasize reconfiguring the software (by setting switches) to use the resources that are available and the most critical resources should be made redundant. If accuracy is important but timeliness is not, then the emphasis should be on reliability rather than availability. In such cases recovery should delay this function until repairs are made. It may be necessary to make provisions so that other functions may be carried out even though the one directly affected by the function is delayed. If both accuracy and timeliness are critical then a high level of redundancy must be provided for hardware, software and data.

Recovery mechanisms must be able to detect failure/abort, record progress of action through stages, manage replicated objects, investigate environment, and remap code and data windows on the fly.

To perform their functions properly, recovery mechanisms must operate both before and after an exception is raised. Before an exception occurs, an action must store information it may need during recovery. After an exception occurs, the recovery handler must limit the consequences of the failure and must construct a new state from which computation can resume.

Our mechanisms must deal with the problems of implicit commits (caused when an action executes an irreversible operation). We must recognize the action aborted and initiate recovery. During recovery we will have to determine how far the action had progressed prior to failure. Suppose, for example, the failure is the result of a hardware error. Recovery could be initiated on a backup machine. If the proper protocols were observed, the recovery handler could recognize where the failed action was just prior to its failure and then restart the action at the appropriate point. Of course, information about what the action did between its last transmission to the backup machine and failure is lost. This is not a problem if lost operations (or their inverses) are idempotent. It may also be possible to observe physical system and other objects to determine whether any of the lost operations

were performed.

EXAMPLE 1: Skeleton of an action

This example illustrates the outline of an action. This example includes details not shown in section 2.2. We have distinguished between the exception handler and the recovery handler. The exception handler is sensitive to the name of the exception (as in Ada). The exception handler may be used to classify results, monitor activity or for error handling. The exception handler uses a case statement to select code appropriate to the exception being handled. The recovery handler is sensitive to the stage in which computation was interrupted by an exception. The recovery handler uses a case statement to select code appropriate to the stage in which the action aborted.

When an exception is raised control transfers to the exception handler. If the appropriate cases have not been provided in the exception handler, the action aborts and control transfers to the recovery handler. The exception handler may reraise an exception visible to the parent action. The exception handler also has the options of terminating the action normally or aborting it. If the exception handler aborts the action, then control passes to the recovery handler. The recovery handler appropriate to the stage at which the exception was first raised is used.

The seperation of the exception handler from the recovery handler is appropriate when the exception handler is being used to clasify results or monitor activity. When errors are being handled however, the recovery often needs to know both where an action aborted and why. Thus, it is possible to nest case statements sensitive to the name of the exception within the recovery handler.

We have also shown the recovery handler seperated into two parts. The **on abort** section is executed immediately when the action aborts. The execution of the **on restart** section is delayed until the action is about to be restarted. We have shown how the **on restart** section can be made sensitive to the circumstances of the restart and the stage in which it is restarted.

Additional information about the reason an action has aborted may be made available through system calls.

This language construct is intended for software design. In some of the examples which follows we have elided or modified portions of the construct to improve the readability of the example. These modifications are explained in conjunction with the examples in which they appear.

```
!data areas global to the action

begin data area 1 [<data area attributes>]
  !declarations go here

end data area 1

begin action 1 [<action attributes>]
  begin data area 2 [<data area attributes>]
    !declarations go here

  end data area 2

  stage 1:

  stage 2:

on exception

  [ some code which is executed for all exceptions ]

   case exceptionType of

    <exception name>:

    <exception name>:

    others:
```

```
  end case

  [ some code which is executed after specific exceptions are handled ]

on abort

  [ some code which is executed on all aborts ]

  case stageAborted of

    stage 1:

    stage 2:

  end case

  [ some code which is executed after handling the stage dependent issues]

on restart

  [ some code which is executed on all restarts ]
  case sourceOfRestart of
    internalRestart:  [some code which is executed on all internal restarts]
                      case stageRestarted of

                      end case
                      [some code which is executed on all internal restarts]

    externalRestart:  [some code which is executed on all internal restarts]
                      case stageRestarted of

                      end case
                      [some code which is executed on all internal restarts]
    end case
```

```
  [ some code which is executed on all restarts]
end action 1
```

6 Mechanisms for detecting faults and initiating recovery

When a fault is encountered, an exception should be raised. When the action recieves the exception, control passes to the exception handler. If the exception handler is not equiped to handle the exception the action aborts and control passes to the recovery handler. This second strategy is the one which should typically be followed for fault related exceptions, since it is often important for the recovery handler to be aware of both the stage in which the action aborted and the reason it aborted.

Some exceptions are are raised by the hardware on which the software is executing. Other exceptions are raised by the system software supporting applications. Still other exceptions are raised within the applications software itself.

Hardware is designed to trap certain types of exceptions such as divide by zero, access violations, and time outs. Software checks may reveal yet other faults arising from errors in data, code or hardware. For example, some hardware faults such as communication failures and crashes on remote sites may not be detected unless special software procedures are used.

Actions may be aborted in several ways. The hardware may fail, it may be aborted by a signal from the system software because of external events, it may be aborted by an internal signal, or it may be aborted because of an internal error which trapped to the system.

When an action aborts, it is necessary to detect the fact that the abort has occurred and to initiate appropriate recovery procedures. The problems of detecting the abort and then initiating recovery must be approached differently for the various kinds of aborts.

If the exception which caused an action to abort was raised by the action itself or by one of its operations, we say the action aborted because of an internal exception. If the exception was raised as a result of activities external to the action, we say the action aborted because of an external exception.

Finally, we can distinguish between explicit and implicit aborts. If an action is aborted in an orderly way by the operating system, we say that an explicit abort has occurred. Only the operating system may explicitly abort an action. The operating system may explicitly abort an action in response to a signal from another action or the action itself. The operating system may also explicitly abort an action for management reasons (e.g.,

resolving deadlock or preempting a resource). Finally the operating system may abort an action because of some illegal behavior (e.g., arithmetic overflow or an access violation) on the part of the action. An implicit abort occurs if the action fails because of events external to the operating system. Hardware and power failures are causes of implicit aborts.

Internal aborts arise when an action aborts itself. This is essentially a transfer of control from the main line of the computation to the appropriate section of the recovery handler.

Explicit, external aborts may be signaled if another action or the operating or run-time systems recognize that an action has faulted, been the victim of a fault in another action, or has possibly encountered an error (in software, hardware, or data) but not yet faulted. For example, if an action recognizes that the data it and other actions have been using is incorrect or inconsistent it may tell the operating system that the other consumers of the data should be aborted and given an opportunity to recover. If an action contains run-time errors such as divide-by-zero, the operating system may abort it using the mechanism for external aborts. If an action reads data from a second action and the second action aborts and changes the shared data during recovery, the first action should be aborted and given an opportunity to recover as well. Again, the external abort mechanism should be used. Recovery can be initiated on another machine after the crash has been detected or it may be initiated when the crashed machine comes back up.

Implicit, external aborts include such events as system crashs. An implicit, external abort is a consequence of events not under the control of the action, run-time system, or operating system. Recovery in the face of this type of abort is more difficult than for other types of aborts.

The handling of implicit, external aborts requires some additional discussion.

Recoverying on a backup machine requires access to replicated data and may occur when the backup machine is unable to communicate with the primary machine. A common strategy is to have two copies of the action executing, one on the backup machine and one on the primary machine. The two copies of the action periodically exchange "I am alive" messages. The primary action may also send data to the backup action. The backup action in this case would be charged with maintaining logs and/or checkpoints.

If the backup action fails to hear from the primary action, the backup action aborts and on recovery assumes the role that had been played by the primary action.

Recoverying on the primary machine is simplier. Actions which were executing when the machine crashed are restarted at an appropriate location in their recovery handler. The recovery handler may attempt to determine whether the action was continued on a backup machine while the primary machine was down. Depending on the answer, the action may be restarted or terminated.

6.1 Mechanisms for detecting faults

The machanisms illustrated in the two examples in this section can be used to determine the reason the action aborted. As was discussed in part I, proper recovery often depends on knowing both the reason an action aborted and where it was in its execution when it aborted. With proper run-time support, either of these mechanism may be used to classify an abort as internal, as explicit and external, or as implicit and external. These mechanisms can be used to establish more detailed diagnosis if required.

EXAMPLE 2: Recovery handlers can be sensitive to both stages and exceptions

This example illustrates a point made in section 5: it is often useful to nest case statements sensitive to the name of the exception within the recovery handler.

```
\begin action

stage 1:

stage 2:

on abort

 case stageAborted of

    stage 1: case exception of
```

```
            preemption:
            resourceUnavailable:
            parentActionFailed:
            usedInvalidData:
            others:
        end case

    stage 2: case exception of
            preemption:
            resourceUnavailable:
            parentActionFailed:
            usedInvalidData:
            others:
        end case
```

EXAMPLE 3: Recovery handlers can use system calls to diagnose circumstances of the abort

To facilitate systems management, actions are encapsulated within objects. A new object is created for each invocation of an action. These "action objects" provide access to the action's code, local data and parent environment. There are also entry points that allow the action, the action manager and other actions to enquire about the action's state. These action management entry points can be used to abort the action and to record information about the circumstances related to its execution, termination (normal or abort). They may also be used during recovery to establish the reason action aborted. This example illustrates the use of object entry points to establish the reason for the abort. The example shows the recovery handler calling an entry point in a resource manager. It also shows the recovery handler calling an entry point into the object encapsulating the action being recovered.

```
\begin action

stage 1:
```

```
stage 2:

on abort

 case stageAborted of

    stage 1: if server@serviceFailure then  <take an action>
             else if self@externalAbort then <take an action>

    stage 2: if objectX@uncommitted then
                 if server@serviceFailure then <take an action>
                 else <take an action>
             else <take and action>
```

6.2 Mechanisms for initiating recovery

As was discussed in part I, it may be necessary during recovery to abort other actions. These other actions may be aborted because they consumed data invalidated during recovery or because their correct execution depends on the successful completion of the action which aborted.

The two examples in this section illustrate different approaches for aborting actions which interact with a faulty action. The first shows an action's recovery handler aborting other actions. The second shows an action aborting itself when it detects that an action on which it depends has failed.

EXAMPLE 4: Aborting other actions during recovery
The recovery handler aborts actions three ways. When it rolled back the uncommitted data area (in stage 1), all the optimistic readers were aborted. These actions recieve the usedInvalidData exception.

It also explicitly aborted the actions whose capabilities were kept in the variables X and

Y. X and Y receive the exception indicated as the message parameter. In stage 2, data area 1 is modified after it was commits. In this situation it is necessary to explicitly abort actions which have accessed the data. the **abort** operation takes a list of the actions which have read the data area as a parameter. The **exclude** parameter indicates actions in the list which should not be aborted. The **message** parameter indicates the exception which is to be sent to the actions targeted to be aborted.

```
data area 1 (optimistic reads allowed)

end data area 1
begin action

stage 1: <use data area 1>;
         commit(data area 1)


stage 2:


on abort
   case stageAborted of

     stage 1: rollback(data area 1)
              abort(X, message=> <an appropriate exception name>);
              abort(Y, message=> <an appropriate exception name>);
              raise(<an appropriate exception name>);

     stage 2: <make some changes to data area 1>
              abort(dataArea1@readers, exclude=> self@parent, message=>usedInvalidData)
              terminate normally

    end case
end action
```

EXAMPLE 5: Noticing that another action has failed

In the previous example an action was responsible for aborting other actions which depended on it. The responsibility can be transfered to the dependent action by using probes. The present example illustrates this idea. Each action consists of a pair of coroutines. In each action, one coroutine in the pair does the work while the other exchanges probes with other actions. If an action fails to hear from the others with whom it is exchanging probes, it assumes they have failed. The action aborts and initiates recovery. Since all of the partners in this arrangement all have the same structure, we show the skeleton for only one of the partners.

```
begin action

stage 1: cobegin
         A: {begin action 2
           loop
            stage 1: X@sendProbe
            stage 2: select
                       receiveProbe(X);
                     or
                       delay <maximum wait>;
                       raise(message => partnerFailed);
                     end select;
            end loop;

           on abort
             case exception of
               partnerFailed: raise(message => partnerFailed);
               other:  raise(message => other);
             end case
           end action 2
         }
```

```
      B:{ begin action 3
         stage 1:
         stage 2:
         on abort

         end action 3
       }
    coend;

    on abort
              case exception of
                partnerFailed: abort(action3 => partnerFailed);
                               abort(X,message => doNotRestart);
                               start an alternative partner;
                               resume(stage 1);
                other: abort(X,messsage => partnerFailed);
                       abort(action3 => selfFailed);
                       raise(internalFailure);
end action
```

7 Recovery activities preceding an abort

If an action is to recover successfully in the event of an abort, it must prepare in advance. In particular, it must ensure that any information which may needed during recovery is properly saved. Some actions are designed to restart on a backup system should the primary system fail. To prepare for this, an action must ensure that the required information is periodically recorded at the backup site. Other actions are simply designed to carry out recovery on the same machine they were executing originally. In this case it is important that the required information is periodically recorded on stable storage.

There are three types of information which must be saved: flags indicating transitions between stages of an action, data checkpointed by a stage, and data logged by a stage. In general, an action may not proceed from one stage to the next until the logs, checkpoints and transition flags have been properly recorded.

Logging A log is an incremental record of transactions. After a system crash, an *undo* operation rolls the log backwards in order to delete a set of transactions. Next, a *redo* operation rolls the log forward in order to redo the operations. There are two types of logging procedures: incremental logging with deferred updates, and incremental logging with immediate updates.

Incremental Logging with Deferred Updates:

> During the execution of a transaction all the write operations are deferred until the transaction partially commits. All updates are recorded on a system-maintained file, called the *log*. When a transaction partially commits, the information on the log associated with the transaction is used in executing the deferred writes. If the system crashes before the transaction completes its execution, or, if the transaction aborts, then the information on the log is simply ignored [15].

After a crash, an *undo* operation need not be executed because the system state has not been modified. After a transaction commits, the *redo* operation rolls the log forward executing all the write operations on the permanent system state.

Incremental Log with Immediate Updates: This method keeps an incremental log of all changes to the system state. All updates are applied directly to the system states and recorded in the log. "If a crash occurs, the information in the log is used in restoring the state of the system to a previous consistent state [15]." In the event of crash recovery, *undo* operation rolls back the log to a previous consistent state. Then the *redo* operation rolls the log forward to a new consistent state.

Checkpoints "When a system failure occurs, it is necessary to consult the log in order to determine those transactions that need to be redone and those that need to be undone [15]." Undoing and redoing a log may be time consuming if the log is long. Therefore, in order to reduce log overhead, "the system periodically performs checkpoints, which require the following sequence of events to take place:

1. Output all log records currently residing in main memory onto stable storage.
2. Output all modified buffer blocks to the disk.
3. Output a log record <**checkpoint**> onto stable storage [15]."

Shadow Paging An alternative to logging is shadow paging.

> Intuitively, the shadow page approach to recovery is to store the shadow page table in nonvolatile storage so that the state of the database prior to the execution of the transaction may be recovered in the event of a crash, or transaction abort. When the transaction commits, the current page table is written to nonvolatile storage. The current page table then becomes the new shadow page table and the next transaction is allowed to begin execution. It is important that the shadow page table be stored in nonvolatile storage since it provides our only means of locating database pages. The current page table may be kept in main memory (volatile storage). We do not care if the current page table is lost in a crash, since the system recovers using the shadow page table [15].

Data Replication The system maintains several identical replicas (copies) of a data item. Each replica is stored on a different node. Replication can enhance the performance

of read operations. It can also increase the availability of data to read transactions. Update transactions, however, involve greater overhead. In most situations, the correctness criteria for data replications is serializability. In order to achieve serializability, a *read set* and a *write set* are defined for each replicated data item. If the sum of the number of replicas in the read set, and the number of replicas in the write set, is greater than the total number of replicas, then serializability is guaranteed.

In order to build a reliable system that uses replication, all of the replicated data items must be consistent. Therefore, an *atomic* transaction called a *commit protocol* that accesses the data items must be constructed that either commits at all replicated sites, or aborts at all replicated sites. The most commonly used commit protocol is the *two-phase commit* protocol. The two-phase commit protocol assumes a central coordinator and a set of nodes each with a copy of the replicated data item.

Phase 1 initiates when the central coordinator forces a *prepare* record to the nonvolatile log. Next the coordinator send a *prepare-to-commit* message to each of the nodes. When a node receives a *prepare-to-commit* message, the node decides whether or not to execute the transaction. If the node decides to execute, then the node forces a *ready* record to the log, and transmits a *ready* message to the central coordinator. If the node decides to abort the transaction, the node forces a *no* record to the log, and transmits an *abort* message to the central coordinator.

Phase 2 begins when the central coordinator receives replies from the nodes. If all of the nodes reply with *ready* messages, then the central coordinator forces a *commit* message to the log and broadcasts a *commit* message to the nodes; otherwise, the central coordinator forces an *abort* message to the log, and transmits an *abort* message to the nodes.

8 Limiting the consequences of the failure

When a fault occurs it is important to limit its impact of system function and performance. We do not want a faulty computation to disrupt other functions which are operating correctly.

The problem of insulating unrelated computations is easily solved: with appropriate hardware support it is possible to isolate applications into separate logical address spaces thereby preventing one application from corrupting the code and data being used.

When applications must exchange information, other measures are required. The simplest measure is to not allow actions to commit before they terminate and to not allow reads of uncommitted data. This is the standard approach. This approach often is not suitable for use in embedded systems. If timing constraints for embedded systems are to be met, we must sometimes allow actions to commit before they terminate (this is another way of saying that under some circumstances it may be desirable for actions to read uncommitted data). This approach is called optimistic concurrency control.

The problem with optimistic concurrency control is that of cascading aborts. We need some sort of "firewall" to prevent the failure of one action from causing the failure of other actions which have read uncommitted data supplied by the failed action. Forward recovery can be used to provide that firewall. Suppose Action A reads uncommitted data written by action B. Action A then makes some data available to action C. Does action C depend on the uncommitted data provided by action B? Only action A knows for sure. If action B aborts and during recovery invalidates the data provided to action A, then A must abort. Forward recovery provides a firewall in that it gives action A an option: action A may determine that the data supplied to action C did not depend on the now invalid values supplied by B. Thus, A can restart itself in an intermediate state and hide from action C the fact that A had aborted and recovered.

Sometimes it will not be enough to simply insure that actions do not use data invalidated by an aborting action. The fact that the failed action did not complete successfully and did not produce the required effects may in itself set up conditions causing subsequent actions to fault.[6] If actions **A** and **B** are executing concurrently and**B** must be preceeded by **A**, the failure of **A** has consequences for the correct execution of **B**. In situations such as this,

[6] this is the phenomenon we have refered to as *cascading faults*.

more active measures can also be used to limit the effects of a failed action. For example, an abort signal can be sent to actions which are active but should not proceed until the failed action has recovered. If it is possible that the propagation of information provided by the failed action will cause anomalous behavior in a subsystem, it may be appropriate to direct the abort signal in a way which causes the subsystem to shut down until recovery is complete. Other possibilities include forcing the subsystem to reconfigure or finding another means for providing the effects which were supposed to have been produced by the failed action.

8.1 Limiting cascading aborts when optimistic reads are allowed

The next two examples illustrate how forward recovery can be used to prevent an optimistic reader from aborting any or all of the actions to which it has supplied data in the event that one of the actions from which it as consumed data aborts.

EXAMPLE 6: A firewall protecting against cascading aborts attributed to optimistic reads

With careful use of locking and recoverable data areas of small granularity, it is often possible to maintain recoverability and availability without allowing actions to read uncommitted data (or, more broadly, to view the effects of uncommitted actions).

In some circumstances availability can be increased if actions are allowed to read uncommitted data. When roll back is the only available recovery technique, this entails a risk of cascading aborts. The risks are compounded if the victims of the cascade of aborts have performed irreversible operations. By regarding the uncommitted data as irreversible once it has been read (or other wise made visible) and by using forward recovery techniques to build a "firewall" against cascading aborts, we can provide a means for inhibiting the cascade. In this example, we assume that the run-time system maintains a record of actions which have been granted access to uncommitted data. If an action aborts and its abort handler performs recovery on a data area which has subsequently been accessed by other actions, then the other actions are aborted.[7] This initiates forward recovery within

[7] In the event that one of these other actions has aborted then its surviving ancestor will be aborted.

these other actions. A casecade of aborts may be avoided in either of two ways. First, the action which has been aborted because it accessed uncommitted data may not itself have permitted yet other actions to access its uncommitted results. Second, even if it had itself permitted access to its uncommitted results, it may be able to recover without affecting the uncommitted results.

The example illustrates how an action can avoid aborting other actions which may have seen its uncommitted results.

```
begin data area 1 (shared access allowed, uncommitted access allowed)

end data area 1

begin data area 2 (shared access allowed, uncommitted access allowed)


end data area 2

begin data area 3 (shared access allowed, access to committed data only)

end data area 3

begin action 1 (executes as a process,
                may access uncommitted data in data areas 1 and 2)

stage 1: readLock(data area 1)
         writeLock(data area 2)
         writeLock(data area 3)
         read from data area 1
         unlock(data area 1)

stage 2: read from and write to data area 2
         unlock(data area 2)

stage 3: do some more stuff
```

```
        unlock(data area 3)

on exception
  abort

on abort
  stage 1: unlock(data area 3)
           unlock(data area 2)
           unlock(data area 1)
           if self@externalAbort() then
             restart stage 1
           else
             exception(internalError) !parent will handle it from here
           end if
  stage 2: if self@externalAbort() then
             rollback(data area 2) !or some other state correction
             abortSubsequentAcesses(data area 2)
             if wasRecovered(data area 1) then
               if <important changes to data area 1> then
                 unlock(data areas 1,2, 3)
                 restart stage 1
               else
                 restart stage 2
               end if
             else if wasRecovered(data area 2) then
                 restart stage 2
             endif
           else
             exception(internalError) !parent will handle it from here
           end if
stage 3: if self@externalAbort() then
           rollback(data area 3) !or some other state correction
           if wasRecovered(data area 1) then
             if <important changes to data area 1> then
```

```
        unlock(data area 3)
        rollback(data area 2) !or some other state correction
        abortSubsequentAcesses(data area 2)
        restart stage 1
      else
        restart stage 3
      end if
    else if wasRecovered(data area 2) then
      restart stage 2
    endif
  else
    exception(internalError) !parent will handle it from here
  end if
```

EXAMPLE 7: A firewall protecting against cascading aborts within a hierarchy of nested actions

```
!This is an example procedure that protects against cascading aborts.  Here
!there are four levels of nestings of actions.  If the fourtch layer
!action aborts, the third layer action notices the aborts and
!computes some forward recovery and returns.  If forward recovery
!cannot be accomplished, then the third layer action aborts and
!the second layer action invokes forward recovery.  If the second
!layer action cannot compute forward recovery, then the top layer
!action handles the the abort.  If the top layer action cannot
!compute forward recovery, then the abort is cascaded.

implementation of object cascade_example
cascade_procedure(parmlist : ...)
begin
   begin action top_level
      stage 1:
         begin action second_level
```

```
            stage 1:
               begin action third_level
                  stage 1:
                      begin action fourth_level
                          stage 1: do some processing
                          on abort raise_exception(except_condition4)
                      end
                  on abort
                      do some forward recovery
                      if exception
                          raise_exception(except_condition3)
               end
             on abort
                do some forward recovery
                if exception
                   raise_exception(except_condition2)
          end
       on abort
       do some forward recovery
         if exception
            raise_exception(except_condition2)
    end
end
```

8.2 Limiting cascading faults

The next five examples illustrate how forward recovery can be used to prevent the failure of one action from causing dependent actions from failing. The first two of these examples show how forward recovery can be used to maintain failure atomicity. The next three examples show how recovery can provide an alternate source for the expected effects.

The last two examples show how the dependent action can be reconfigured to remove the dependence on the failed action. The first of these shows reconfiguration within nested actions and the second shows reconfiguration across independent threads of control.

8.2.1 Maintaining failure atomicity

EXAMPLE 8: systems which ignore redundant commands

Assume **g** and and j exhibit strong precedence. As defined in section 2.2.2, strong precedence exists between two operations A_1 and A_2 if

1. A_1 must precede A_2 if A_2 is to occur at all, and
2. A_2 must follow A_1 if A_1 is to occur at all.

i is an irreversible operation. Assume that **g*** (the inverse of **g**), h* (the inverse of h), and j are idempotent. Operation i is not idempotent but operations exist to establish the status of the physical system on which i operates.

Recall that an idempotent operation has no effect if the state the operation is to bring about already exists. For example, turning off a light has no effect if the light is already off.

```
begin action

stage 1:  g(x);

stage 2: h(x);

stage 3: i(x);  !this is an irreversible operation

stage 4: j(x);

on abort

stage 1:  g*(X);              !undo any effects of g and give up
          raiseException(noEffect);
stage 2:  h*(X);              !h is optional so undo it and go on to the
          resume(stage 3);    !next stage
stage 3: if i_notPerformed(x) then
```

```
          resume(stage 4);
        else if i_partiallyPerformed(x) then
          k(x);
          resume(stage 3);
        else
          resume(stage 3);
        end if;
stage 4: resume(stage 4);

end action
```

EXAMPLE 9: Failure atomicity in resource management

Checking a resource out of a pool is one type of irreversible action. Recovery may be used to ensure that the resource is returned.

```
data area is visible to actions other than the one shown

    begin dataArea 1
       X, Y: capabilities for a resource
       Z: data of some kind
     end dataArea 1

     begin Action 1
       checkPoint dataArea 1
       stage 1: X := server@obtainResource()

       stage 2: X@initialize()
                while (condition) do
                   Z := something
                  X@write(Z)
                end while

      stage 3: X@cleanup()
                server@return(X)
```

```
    on abort
      stage 1: server@parent@serviceFailure(server)
               rollback dataArea 1
               raise exception(notStarted)
      stage 2: some operations on X to make it consistent
               X@cleanup()
               server@return(X)
               raise exception(incomplete)
      stage 3: server@parent@serviceFailure(server)
                terminate normally
end action
```

8.2.2 Providing an alternative source for the expected effects

The first example shows how an action can provide the alternate source in the event it fails. The second and third examples show how a backup action at another site can use probes to detect that the first action has failed and take over for it.

EXAMPLE 10: Maintaining failure atomicity by providing an alternate source of required effects

Assume that other actions require the effects of operation c. If c cannot be executed then d will provide some of the results. Operation d will also set some flags that to indicate that the action executed with reduced functionality. Other actions will see the flags and be will be able to adjust themselves to the reduced functionality of the first action.

The use of flags can be avoided by remapping the code window. This use of code windows is discussed in section 5.2. in section 5.2.

```
begin action

stage 1: a(x);
```

```
stage 2: b(x);

stage 3: c(x);

on abort
  setFlags(F);
  d(x);
  terminate normally
end action


! the dependent action

begin action

 if not flagsSet(F) then do one thing
 else do another thing

end action
```

EXAMPLE 11: a second example using probes
This example shows how probes can be used to restart a computation on another machine. Note that the storing of information needed during recovery is handled by the action. Information stored can include the current stages of the coroutines. This information can be used to restart the action on another machine in an intermediate state. action 1 ersion:

```
procedure probe()
  broadast id
  if recieve an id from a higher priority replica then abort
end
```

```
begin action

stage 1:  active := false;
stage 2:  loop        !dormant version
           eaves drop listening for activity from higher priority replicas
           store data recieved from active replica
           abort if no activity heard
          end loop

stage 3:  cobegin          !activeVersion
          probe: { loop
                     exchange probes
                     if probe from higher priority replica recieved then
                     abort;
                     if probe from lower priority replica recieved then
                     signal it to abort
                   }
           replicateData: { loop
                              send important data to dormant replicas
                              keep replicas apprised of progress
                            end}
           doWork: { code for doing the work required of this action}
          coend

on abort
     stage 1: resume stage 1;
     stage 2: attempt to verify the active version is dead
              do some work to construct a consistent state
              active version := true;
              resume stage 3
     stage 3: send out any information other replicas may need
              active version := false;
              resume stage 2
```

EXAMPLE 12: Probes

```
!  This example is an implementation of the procedure stat_check
!  defined in example 1.  stat_check returns true iff the primary
!  is marked as operational and returns probes, or the backup
!  is marked as operational and returns its probes.
!
!  This action uses an additional language feature called a ''timed action''.
!  A timed action aborts if it does not terminate before its timed
!  period expires.


stat_check(index : index_type)
begin
   begin action chk
      stage 1: assert(resrc_tbl[index].primary->status)
      stage 2: begin timed action (TIME_OUT_CONST)
                  send_probe(ARE_YOU_UP,resrc_tbl[index].primary->addr)
                  receive_probe(val)
  assert(val = up)
      return true

      on abort

      CASE stage OF
        stage 1 :
           ! the assertion failed
           if (resrc_tbl[index].backup->status)
               then
                   swap(resrc_tbl[index].primary,resrc_tbl[index].backup)
                   resume(2)
```

```
        else
                    raise_exception(operation_not_available)
         stage 2 :
            if (exception = timed_out) or (val = down)
               then
                   if (resrc_tbl[index].backup->status)
                     then
                        swap(resrc_tbl[index].primary,resrc_tbl[index].backup)
                        resume(2)
              else
                        raise_exception(operation_not_available)
               else
  raise_exception(operation_not_available)
                   !give up
      end;
end;
```

8.2.3 Using forward recovery to reconfigure the software

EXAMPLE 13: processes can be killed to force reconfiguration

```
begin action

stage 1: a(x);

stage 2: b(x);

stage 3: c(x);

on abort
  abort(Action2, message => reduceServiceLevel);
```

```
  d(x);
  terminate normally
end action


! the dependent action

begin action

stage 1: if not flagsSet(F) then do one thing
         else do another thing

on abort
   case exception of
    reduceServiceLevel:  rollback;
                         setFlags(F);
                         resume(stage 1);
end action
```

8.2.4 incremental recovery over several levels of nesting

EXAMPLE 14: incremental recovery over several levels of nesting

```
begin action

    begin action

        begin action


        on abort
           <do some partial recovery>
```

```
            raise exception
         end action


      on abort
           <do some more recovery>
            raise exception

      end action


on abort
      <do some more recovery>
      terminate normally
end action
```

9 Repairing the computation

9.1 Using redundant data

The state of a computation can contain invalid or inconsistent data and by the time the error causes a fault or is otherwise detected, it may no longer be possible to detect the source of the error. It is often possible, indeed it may be sufficient, to correct the error and allow computation to resume. It is also reasonable to examine critical data for correctness and consistency before a fault occurs. This could be done by a coprocessor or by the main event loop during otherwise idle moments. Such a precautionary measure is reasonable to protect against commutative errors. For example, over a period of time the software may become increasingly uncertain as to the location of a particular part under its control. If the system become idle, it may be appropriate the software to put the system through a recalibration cycle.

four general strategies are available for correcting erroneous data and/or recovering missing data:

- measure physical system and use results to determine correct value
- replace incorrect data with backup data.
- compute a replacement value using other values in the current state and perhaps values which have been checkpointed. This strategy is often feasible because the state contains redundant information. This last approach regards the state of the physical system as one more repository for redundant data.

When an action fails, information regarding its state may be lost if it was not flushed to stable storage prior to the failure. This may result in an inconsistency (the stable version may contain current values for some variables but not for other variables use to compute those new values). These problems may be handled using the techniques described above. The emphasis here is on recovering missing information regarding the occurrence or non-occurrence of irreversible actions.

The techniques described here can also be used to record progress from one stage to another within an action.

If the intention is not recorded, the operation was not invoked. If the completion is recorded, then the operation was invoked. If the intention is recorded and the completion not, then the situation is indeterminate. Several strategies are possible in this situation. If the operation is idemnopotent, it may be performed again to ensure the operation has been completed. If the inverse is idempotent, it may be performed again to ensure the operation is undone. If this strategy is inappropriate, then the subsystem must attempt to establish whether the operation occurred. This can be done by checking the status of devices attached to the system. With proper foresight, system designers will have provided a connection to the system affected by the operation which allows the question to be answered. Redundant communication channels are also a possibility. The backup machine may eavesdrop on the conversation between the primary machine and the physical system. Even if the primary machine failed before it made a record that the irreversible operation completed, the back up machine will have observed that the appropriate commands were issued to the system. Thus, the backup machine can infer that the operation was performed.

Similar provisions can be made even if the action is of low criticality and recovery can wait until the primary machine is brought back up.

EXAMPLE 15: Saving information in the global environment
This example illustrates the use of our forward recovery constructs to propogate information into the global environment. The parent action will then use that information to select an appropriate continuation.

```
begin data area 1
   server: capability
   finished: boolean := false
   numberDone := 0
end data area 1

begin action 1
  begin data area 2
    X: capability
    Z: data of some kind
    c: integer := 0
```

```
  end data area 2

  begin
    stage 1: X := server@obtainResource()
             X@initialize()

    stage 2: while <condition> do
                checkpoint data area 2
                c := c + 1
                Z := <some expression>
                X@useResource(Z)
                commit data area 2
             end while

    stage 3: X@cleanup()
             server@return(X)
on exception
  abort

on abort
  begin data area 3
    reason: (serverFailure,resourceFailure, other)
    action: (raiseException, normalTermination)
  end data area 3

  if server@serviceFailure(server) then
    reason := serverFailure
  else if server@serviceFailure(X)
    reason := resourceFailure
    server@return(X)
  else reason := other
  end if

  case stageAborted of
```

```
   stage1:  action:= raiseException

   stage 2:  if Z <> X@lastValue() then
                rollback data area 2
             end if;
             action := raiseException

  stage 3: action := terminateNormally
  end case

  numberDone := c
  if action = raiseException then
    finished := false
    exception(reason)
  else
    finished := true
    terminate normally
  end if

end action 1
```

9.2 Using redundant code

Redundant code can be exploited during recovery in several ways. This simplest is for the recovery handler to replace the code containing the error with another version not containing the error. This can be done be remapping the code window for the action which aborted. The new code may require a different presentation of the state information; consequently, it may also be necessary to remap the data window as well. The contents of the new data window will be calculated by the recovery handler using the contents of the old data window.

A common strategy will be to cut over not to an equivalent section of code but to one with reduced functionality. The simpler version will operate until repairs can be made to the primary version. The primary version can be reinstalled by using the cutover strategy.

A rolling cutover is also possible. When the old version of an action attempts an

operation not supported in the version with reduced functionality it aborts. During recovery the appropriate backup version with reduced functionality is swapped in.

The cutover stays in place until repairs are made and the patched, full function version is ready to be reinstalled. Cutting over to the repaired, full function version can be achieved in the same way as was the cutover to the reduced level of functionality.

A cutover is triggered by errors within a module. Generally, this is a way of handling software errors. That the error is detected suggests either that an error trapped to the system while the module was executing or that a data error was traced back to the module (suggesting an acceptance test either in the module or on the consumer of the module's data)

A similar strategy may be used to obtain a correct computation given two partially correct versions. If the acceptance test on one version fails, the recovery manager would cut over to the other and redo the computation. When states are encountered which cause the second version to produced unacceptable results, it aborts and the recovery manager cuts back to the first version. A strategy such as this would be plausable provided the two versions were developed independently and simple acceptance tests could be provided to check the results.

EXAMPLE 16: Retry or terminate
This example illustrates the use of our forward recovery constructs to declare an action which clean up its state and either retrys itself or terminates by raising an exception visible to its parent.

```
begin data area 1
   server: capability
end data area 1

begin action 1
  begin data area 2
    X: capability
    Z: data of some kind
  end data area 2
```

```
  begin
    stage 1: X := server@obtainResource()
             X@initialize()

    stage 2: while <condition> do
                checkpoint data area 2
                Z := <some expression>
                X@useResource(Z)
                commit data area 2
             end while

    stage 3: X@cleanup()
             server@return(X)
on exception
  abort

on abort
  case stageAborted of
   stage 1: if server@serviceFailure(server) then
              exception(serverFailure)
            else if server@serviceFailure(X) then
              server@return(X)
              retry self
            else
              rollback data area 2
              retry self
            end if
   stage 2: if server@serviceFailure(X) then
              server@return(X)
              exception(actionIncomplete)
            else if Z = X@lastValue() then
              commit data area 2
              resume stage 2
            else
```

```
            rollback data area 2  !to a state consistent with the begining of the loop
            resume stage 2
          end if

  stage 3: if server@serviceFailure(X) then
            server@return(X)
          else
            X@cleanup()
            server@return(X)
          endif
          terminate normally
 end action 1

end action 1
```

The serviceFailure call begins diagnostic routines and may result in corrective action within the server or resource. This ensures that continuity of service is maintained.

Note that the X@useResource(Z) is treated as a potentially irreversible action. The abort handler uses the operation X@lastValue() to determine whether it was invoked just before the the abort occured. the result of the lastValue operation will determine how recovery will proceed.

EXAMPLE 17: Restart a successor action

This example illustrates how the abort handler may attempt to complete an action by starting a successor action. The abort handler copies data into the global environment. The restart handler will copy that data into the local environment of the successor action. This is an example of an external restart.

In this case the successor action is defined within the parent action and has access to the data which is propogated into the global environment.

This example also illustrates a different strategy for maintaining continuity of service from the server.

```
begin data area 1
   server: capability
   finished: boolean := false
   numberDone := 0
   resource: capability := null
end data area 1

begin action 1
  begin data area 2
    X: capability
    Z: data of some kind
    c: integer := 0
  end data area 2

  begin
    stage 1: X := server@obtainResource()
             X@initialize()

    stage 2: while <condition> do
               checkpoint data area 2
               c := c + 1
               Z := <some expression>
               X@useResource(Z)
               commit data area 2
             end while

    stage 3: X@cleanup()
             server@return(X)
on exception
  abort

on abort
  begin data area 3
    stage: (stage1, stage2, stage3)
```

```
end data area 3

if server@serviceFailure(server) then
   server := server@successorServer()
else if server@serviceFailure(X) then
   server@return(X)
   X := server@obtainResource()
end if

case stageAborted of
 stage1:  stage := stage1

 stage 2:  if Z <> X@lastValue() then
              rollback data area 2
           end if;
           stage := stage2

stage 3: stage := stage3
end case

if stage = stage3 then
   if not server@confirmReturn(X) then
      server@return(X)
   end if
   terminate normally
else
  !propogate state into the global environment
  resource := X
  numberDone := c
  if stage = stage1 then
    restart using self@fparent@alternative()

  else if stage = stage2 then
    restart using self@fparent@alternative() in stage 2
```

```
    endif
  endif
end action 1
```

EXAMPLE 18: Remapping code windows

This example illustrates how the abort handler may attempt to complete an action by mapping code for the successor action into the code window of the action which is aborting. The abort handler may also remap the window containing the restart handler. On restart, the local data areas and the abort handler may also be remapped.

This is an example of an internal restart. The restarted action inherits the local environment of the action which it replaces.

```
begin data area 1
   server: capability
   finished: boolean := false
   numberDone := 0
   resource: capability := null
end data area 1

begin action 1
  begin data area 2
    X: capability
    Z: data of some kind
    c: integer := 0
  end data area 2

  begin
    stage 1: X := server@obtainResource()
             X@initialize()

    stage 2: while <condition> do
                checkpoint data area 2
                c := c + 1
```

```
                Z := <some expression>
                X@useResource(Z)
                commit data area 2
             end while

    stage 3: X@cleanup()
             server@return(X)
on exception
  abort

on abort
  begin data area 3
    stage: (stage1, stage2, stage3)
  end data area 3

  if server@serviceFailure(server) then
     server := server@successorServer()
  else if server@serviceFailure(X) then
     server@return(X)
     X := server@obtainResource()
  end if

  case stageAborted of
   stage1:  stage := stage1

   stage 2:  if Z <> X@lastValue() then
                rollback data area 2
             end if;
             stage := stage2

  stage 3: stage := stage3
  end case

  if stage = stage3 then
```

```
      if not server@confirmReturn(X) then
         server@return(X)
      end if
      terminate normally
   else
     !propogate state into the global environment
     resource := X
     numberDone := c
     if stage = stage1 then
       remap codeWindow using <information needed to complete the remapping>
       remap restartWindow using <information needed to complete the remapping>

     else if stage = stage2 then
       remap codeWindow using <information needed to complete the remapping>
       remap restartWindow using <information needed to complete the remapping>
       restart in stage 2
     endif
   endif
end action 1
```

EXAMPLE 19: Redundant software to provide a fault tolerant system service

```
!  Every resource has a backup.  If the primary is not operational, then
!  the backup is invoked.  If neither is operational, the resource
!  cannot be used.
!
!  This example uses a special keyword ''RESUME(n)''.  This keyword
!  overrides the exception handler's normal protocol by resuming
!  the action in stage n.  Note that if the RESUME keyword is not
!  used the exception handler terminates when an exception is raised.

implementation of object resource_mgr
type
```

```
   resrc_entry
      primary : pointer to resource
      backup  : pointer to resource
   end;

begin data_area resrc
   resrc_tbl : array (index) of resrc_entry
end data_area resrc

resrc_invoke(index : index_type; op : operation; data : data)
begin
   begin action rec
      stage 1: stat_check(index)
      stage 2: invoke(index,op,data)

      on abort

      CASE stage OF
        stage 1 :
           if not resrc_tbl(resrc_tbl[index].primary->status)
              then if not resrc_tbl[index]backup->status
                 then raise_exception(operation not available)
                 else begin
                    swap(resrc_tbl[index].primary,resrc_tbl[index].backup
                    resume(2)
                 end
              else raise_exception(status_check_exception)
       stage 2:
           if not resrc_tbl[index].backup->status
              then raise_exception(operation not available)
              else begin
                 swap(resrc_tbl[index].primary,resrc_tbl[index].backup)
                 resume(2) !retry using backup
              end
```

```
end
```

EXAMPLE 20: Redundant subsystems

```
implementation of object redundant_software
! One method used to implement fault tolerance is to use a set of
! systems that are all supposed to do the same thing.  Each system
! is designed and implemented by a different team of engineers.
! In this example, when a system notices that something is wrong,
! the system voluntarily gives up control to a top-level controller.
! The top-level controller invokes another system

redundant1(state : state_type)
begin
   begin action r1
      stage1 : do some processing
      on abort
         raise_exception(state_information)
   end
end

...

redundantn(state : state_type)
begin
   begin action r1
      stage1 : do some processing
      on abort
         raise_exception(state_information)
   end
end

top_level_controller
```

```
begin
   begin action control
   begin data area ctrl
      bad_list : array[1..n] of boolean
      state_information
      procedure_name
   end data area

   stage 1 : procedure_name := pickproc(bad_list) !try a new system.  If no
                                                  !systems are left give up
   stage 2 : invoke_some_controller(procedure_name)

   on abort
   case exception of
      1 : raise_exception(no_system_works)
      2 : bad_list(state_information.redundant_number) := FALSE
          resume(1)
   end
end
```

EXAMPLE 21: Irreversible actions: robotics

```
Implementation of object robot_arm

begin data area
  arm_position : pos_tensor
end data area

! The initialize procedure figures out the robot's configuration
! by invoking low-level sensors associated with each joint.
! Each low-level sensor returns a variable describing its joint's
! angle, rotation, or extension (whichever is appropriate).
! The initialize procedure accepts the sensor's values to compute
```

```
! tensors that describe the exact postion of the end of the robot's
! arm.  Initialize is invoked as part of the recovery procedure when
! the robot's control system faults.
!
function initialize : position : pos_tensor
begin
   begin action init
      stage 1 : invoke appropriate sensors
      stage 2 : compute tensors

      on abort

      case staged abort of
      stage 1 : raise_exception(joint_broken(joint_number))
      stage 2 : raise_exception(invalid sensor values)
   end
end

!Move uses the move_resumption procedure to recompute the arm's position
!and plan whenever necessary.  The recovery procedure always implements
!exactly one retry when needed.
!
procedure move(IN to_place : pos_tensor)
begin
    begin action mv
       stage 1 : validate(arm_position)
       stage 2 : validate(to_place)
       stage 3 : compute_movement_plan(arm_position,to_place)
       stage 4 : execute_movement_plan(arm_position,to_place)

       on abort

       case staged abort of
       stage 1 : begin
```

```
   arm_position := initialize
   if exception
then raise_exception(exception)
          else resume(2)
   fi
 end
       stage 2 : raise_exception(to_place_invalid)
       stage 3 : begin
   move_resumption(arm_position,to_place)
   if exception
then raise_exception(exception)
else resume(4)
   fi
       stage 4 : begin
   move_resumption(arm_position,to_place)
   if exception
then raise_exception(exception)
else execute_movement_plan(arm_position,to_place)
if exception !avoid infinite retry
then raise_exception(exception)
fi
   fi
end
end

procedure move_resumption(to_place)
begin
   begin action mv_res
       stage 1 : arm_position := initialize
       stage 2 : compute_movement_plan(arm_position,to_place)

       case staged abort of
       stage 1 : raise_exception(unable_to_initialize)
       stage 2 : raise_exception(unable_to_plan)
```

```
   end
end


!idem_potent_move is the same as move except there is no need to
!to call the initialization procedure.  In this case, the robot
!simply moves its arm to a location.
procedure idem_potent_move(IN to_place : pos_tensor)
begin
    begin action mv
       stage 1 : execute_idempotent_movement_plan(to_place)

       on abort

       case staged abort of
       stage 1 : begin
                    execute_idempotent_movement_plan(to_place)
   if exception
then raise_exception(movement_unavailable)
   fi
end
end

! reduce_move is a reduced functionality move.  Here, when an execution
! fails, the movement is reexecuted with reduced functionality (possibly
! lower performance or with a decreased set of arm movements.
!
procedure reduce_move(IN to_place : pos_tensor)
begin
    begin action mv
       stage 1 : validate(arm_position)
       stage 2 : validate(to_place)
       stage 3 : compute_movement_plan(arm_position,to_place)
       stage 4 : execute_movement_plan(arm_position,to_place)
```

```
        on abort

        case staged abort of
        stage 1 : begin
    arm_position := initialize
    if exception
then raise_exception(exception)
          else resume(2)
    fi
 end
        stage 2 : raise_exception(to_place_invalid)
        stage 3 : begin
    move_reduced_resumption(arm_position,to_place)
    if exception
then raise_exception(exception)
else resume(4)
    fi
        stage 4 : begin
    move_reduced_resumption(arm_position,to_place)
    if exception
then raise_exception(exception)
else execute_movement_plan(arm_position,to_place)
if exception !avoid infinite retry
then raise_exception(exception)
fi
    fi
end
end

!move_reduced_respumption returns only a subset of the plans
!as in move_resumption

procedure move_reduced_resumption(to_place)
```

```
begin
   begin action mv_res
       stage 1 : arm_position := initialize
       stage 2 : compute_reduced_movement_plan(arm_position,to_place)

       case staged abort of
       stage 1 : raise_exception(unable_to_initialize)
       stage 2 : raise_exception(unable_to_plan)
   end
end
```

9.3 Using redundant hardware

When designing a fault tolerant system, it is possible to provide for the continuation of critical services in the event of hardware failure by making provisions to move computation to a backup machine. The movement of the computation may be directed from either the primary or the back up machine. If the primary machine fails completely, then the movement of the computation must be directed from the backup machine. If only a portion of the primary machine fails, then the primary machine itself may direct the transfer of the computation.

For some critical applications, more than one backup machine may be provided. In some circumstances it may be desirable to allow the possible backup machines to elect one of their number as the machine which will assume the responsibilities of the failed machine. In other circumstances the backup machines may not be able to communicate with each other. In such a situation it is necessary for the machines to have an indirect means of detecting the presense of other machines attempting to perform the backup function. This could be done via redundant, low bandwidth communications channels or by making inferences from state changes in the physical system being controlled. It would also be possible for a processor attached to the physical system to select the backup system from which the physical system will accept control signals.

In order to construct highly available systems, each hardware and software component must be duplicated.

At minimum the system requires two processors. There may have to be two paths connecting the processors, and it is desirable to have at least two paths from the processors to the database, [operating system, application program, etc.] that is, two I/O subsystems consisting of a channel (I/O processor), controller, and disk drives. The disk controllers must be multiported, so that they may be connected to more than one processor [14].

EXAMPLE 22: Hardware Redundancy

```
! The swithover object controls a group of processors.  If all of the
! processors are down then switchover aborts.  Otherwise, switchover
! switches to a backup and continues

implementation of object switchover
type
   proc_entry =
       status variables
       control structures
       ...
   end
begin data area
   processor_list : array (index) of proc_entry
   down_list : array(index) of boolean

! the control loop that controls a particular processor
control(processor : proc_entry)
begin
begin action ctrl
   loop
      control ''processor''
      ...
   end loop
```

```
   on abort
   raise_exception(processor_unavailable(state_info))
       ! state info represents some portion of the state that is
       ! returned to the parent
end
end

!command_control raises an exception if no processors are available.
!otherwise the procedure switches to some backup processor.
!The state is saved and passed on to the backup
!
command_control
begin data area
   state_info : state_information_type
end data area

begin
begin action comm_ctrl
   stage 1: choose some processor, i, not marked on down list
   stage 2: initialize(i,state_info)
   stage 3: control(i)

   on abort
   case exception of
      1 : raise_exception(processors_unavailable)
      2,3 : down_list(i) := FALSE
            resume(1) !with updated state_info altered by the child
    end
end
end

end !object
```

EXAMPLE 23: Two processes on two machines

```
Master

action 1

control the device and communicate with the slave through rpcs on
objects

on abort
  abort because contact lost with slave machine.  Either the slave
  machine crashed or the network partitioned.

  assumes it has effective contact with the command and sensor registers.
  one of the command register's is a deadman's switch.  If the partition
  cut the link to the c&s registers or if this machine crashes, then
  the switch closes.  If the slave still exists and the switch closes
  it attempts to control in an autonomous mode.


end action 1


Slave


  coupled with the master if couple broken, then in autonomous mode.
  effective or ineffective depending on the switch.  no need to actually
  know.
```

9.4 A Detailed discussion of hardware processor redundancy for fault tolerance

In this section we present a brief overview of systems that exhibit hardware provided fault recovery.

Tandem NonStop System

> Each processor module consists of a central processing unit (CPU), memory, interface to an interprocessor bus system Called Dynabus, and an I/O channel. Each of the I/O controllers is connected to two processors via its dual-port arrangement, and each processor is connected to all other processors via a dual Dynabus. Further, ..., each processor is connected to a pair of disk controllers, which in turn maintain a string of up to four pairs of (optionally) mirrored disk drives...

The Tandem system is designed to survive any single point failure. Also, any repair of a single point failure can be accomplished without affecting the rest of the system.

> In order to detect a processor failure in the Tandem system, each processor broadcasts an "I-AM-ALIVE" message every 1 second and checks for an "I-AM-ALIVE" message from every other processor every 2 seconds [3]. If a processor decides that another processor has failed to send the "I- AM-ALIVE" message, it initiates recovery actions...

Stratus/32 Continuous Processing System

> The Stratus/32 Continuous Processing System [13,21] consists of 1-32 Processing Modules, where each Processing Module consists of duplicated CPU, memory, controller, and I/O... The memory may be configured to be redundant or nonredundant, as the two memory subsystems are not paired with the two CPUs. In a redundant configuration, the CPUs read and write to both memory subsystems simultaneously; in a nonredundant configuration, each memory subsystem becomes an independent unit and the memory capacity is doubled. Each

> Processing Module has duplicated power supplies. The Processing Modules are connected through a dual-bus system called the StrataLINK [14].

All hardware malfunctions are reported to maintenance software. The maintenance software determines the cause and nature of the malfunction and directs recovery. In a redundant configuration, Stratus reduces the probability of a hardware- induced error by comparing the results of computation from duplicated hardware components.

System D Prototype

> System D is a distributed transaction processing system designed and prototyped at IBM Research, San Jose, as a vehicle for research into availability and incremental growth of a locally distributed network of computers [1]. The system was implemented on a network of Series/1 minicomputers interconnected with an insertion ring... [14]

Hardware error diagnosis and recovery is implemented by a software subsystem called the Resource Manager (RM).

> The basic premise of its [RM] design is that the time-out mechanism detects all failures, including deadlock, agent or module crash, communication medium failure, or processor failure... The Resource Manager attempts to bring down and restart failed agents, while normal service requests are handled by other agents. In the event of a processor failure, agents are brought up in the backup processor and all transactions in progress are aborted [14].

10 Action-based programming for distributed systems

10.1 A highly available distributed calendar

Our example illustrates a distributed consensus protocol implemented on a fully connected point-to-point network used in a highly available distributed calendar. At the applications level, a user is presented with the following operations: insert, delete, and query. The consensus protocol is two phased and is managed by a central coordinator. The central coordinator requires universal consensus for any calendar update. Consensus is not required for the calendar read operation.

The example illustrates two irreversible actions. An irreversible action is an action that cannot be rolled back after the action initiated. The two irreversible actions are multicast and consensus. Multicast sends an identical message to every machine in a group. Consensus is an applications level action that implements the calendar insert and delete operations. Multicast is irreversible because multicast may only be implemented by a set of point to point send operations. The multicast operation succeeds if and only if every point-to-point send operation succeeds. The action is irreversible because each atomic send operation is irreversible. The second irreversible action, consensus, uses multicast as a nested action. Consensus succeeds if and only if every machine reaches agreement. Consensus is implemented as a two-phase consensus protocol. A central coordinator first multicasts a precommit message, and after receiving a positive reply from all machines, multicasts a commit message. The consensus action is irreversible because each multicast is an irreversible atomic action.

The multicast action is implemented in three stages: initialization, processing, commit. Initialization is the recoverable stage of the action. The initialization stage allocates the data structures used by the action and may be recovered by rolling back. The second stage is the irreversible component of the action. After each transmission, the action checkpoints its progress. Recovery is implemented by multicasting an abort message to each to each destination indicated by a checkpointed list of destinations. The checkpointed list of destinations indicates the destinations to which a precommit message was sent.

The consensus action succeeds if and only all machines reach a consensus. The consensus action is irreversible because every multicast is irreversible. Consensus is implemented in three stages: initialization, precommit, and commit. The initialization allocates the action's

data structures, and may be recovered by rolling back. Precommit broadcasts a message using the multicast action, and receives a reply from each machine. If consensus is not reached, the second stage exception handler is invoked. The exception handler multicasts an abort message. If consensus is reached, the third stage begins execution. The third stage commits the action. A third stage exception is raised on a disk error.

10.2 A walk through of the example

The calendar is a list of records of type *msg_type*. The date and time fields of a *msg_type* record are used to compute a unique key. The key is used to lookup an individual record in the calendar.

```
msg_type = record (export)
    date : pending
    time : pending
    key : key_type
    type : pending
    node : pending
    data : string
end
begin data area calendar_state
    log : log_object
end data area calendar_state
```

insert

The *insert* procedure is the central coordinating procedure of the calendar. *Insert* implements the two phase consensus algorithm. *Insert* is implemented as an action.

stage 1

Interact with the user through the user interface to obtain a calendar entry. Read the old entry from the calendar (section 10.2) into a log record data structure, and commit the old

calendar entry to the log. Any exception raised by the commit action (section 10.2) is a fatal error. The log entry has the type "precommit". Any query (section 10.2) on a date/time slot marked by a log entry with the type "precommit" gets the value of the log entry as opposed to the value of the calendar entry. Therefore, the user view of the calendar does not change until after the precommit portion of the protocol completes. The new calendar entry is then written to the calendar (an exception is fatal).

If this node crashes when a log entry has the value "precommit", the recovery object invokes the procedure *recover_precom* (section 10.2). *Recover_precom* operates by broadcasting an abort message.

stage 2

Broadcast a precommit message to every node. We assume our network topology is point-to-point and fully connected. The broadcast action is *irreversable* because some, but not all, of the nodes may receive the broadcasted message. In this case (a stage 2 exception) the exception handler invokes *no_precommit* (section 10.2) which sends an abort message. If stage 2 procedes normally, all receiving nodes execute the *rec_insert_precommit* (section 10.2) procedure. Otherwise, if an abort message is sent, all receiving nodes execute the *ab_ins_pre* procedure (section 10.2).

stage 3

Receive a message from every node. If all the messages have the value "yes", then commit "precommit good" to the log. At this point, a query will reflect the updated calendar entry. Also, if this node crashes, the recovery object will invoke *recover_ins_com* (section 10.2). *Recover_ins_com* operates by rolling the action forward.

stage 4

Broadcast an "insert commit" message. Any exception is considered fatal. All receiving nodes execute the *rec_insert_commit* (section 10.2). If some node does not receive the "insert commit" message, then the receiving node will not clear the calendar entry from its log. This case will be noticed by the recovery procedure *recov_badcommit* (section 10.2).

stage 5

Receive a "yes" message from every node. Once stage 5 completes, this node is aware that every other node has committed the updated entry to its remote instance of the calendar object.

stage 6

Clear the entry from the log. Any exception is fatal.

code

```
procedure insert
begin data area ins
   log_rec : log_rec_type
end data area ins
begin action insert_action
   stage 1 :
      insertIO(logrec.logmsg)
      logrec.logmsg.key := compute_key(logrec.logmsg)
      read_logrec(logrec.logmsg.key,logrec) !read from calendar
      logrec.logtype := "precommit"
      log@computekey(logrec)
      log@log_commit(logrec)
      write_cal(msg)
   stage 2 :
      logrec.logmsg.type := "insert precommit"
      communicate@synch_bcast(myname,logrec.logmsg)
   stage 3 :
      communicate@synch_recv_all_yes(myname)
      log@commit_type(logrec.logkey,"precommit good")
   stage 4 :
      msg.type := "insert commit"
      communicate@synch_bcast(myname,logrec.logmsg)
```

```
    stage 5 :
        communicate@synch_recv_all_yes(myname)
    stage 6 :
        log@clear(logrec.logkey)
    on abort
    CASE stage OF
    stage 1 : if (exception = key_not_computed)
        then raiseException(invalid_date_or_time)
        else begin
            log@clear(logrec.logkey)
            raiseException(fataldisk_error)
        end
    stage 2 : no_precommit(exception,logrec,msg)
        parm := exception !raised by no_precommit if except exists
    stage 3 : no_agreement(logrec.logkey,exception)
        raiseException(insert_unavailable(exception))
    stage 4 : raiseException(fatal_commit_comm_error)
    stage 5 : no_commit(logrec.logkey)
        if (exception=fatal_commit_error) raiseException(exception)
            raiseException(remote_state_undetermined)
            !If the exception is from no_commit raise fatal_commit_error
            ! else raise an exception (not necessarily an error)
    stage 6 : raiseException(fatal_disk_error)
end action
end
```

no_precommit

no_precommit is called by the stage 2 exception handler of *insert* (section 10.2). The purpose of *no_precommit* is to abort a "precommit" message sent by *insert* in stage 2. If *insert* raised the exception "port_unavailable", then *no_precommit* raises a fatal exception. Otherwise, *no_precommit* broadcasts an "abort insert precommit" message, and clears the log. All receiving nodes invoke *ab_ins_pre* (section 10.2) when the abort message is received.

code

```
procedure no_precommit(exception : IN exception_type;
        logrec : IN log_rec_type;
        msg : IN msg_type)
begin
    if (exception = port_unavailable)
        then raiseException(fatal_port_unavailable)
        else begin
            log@commit_type(logrec.logkey,"precommit bad")
            msg.type := "abort insert precommit"
            communicate@synch_bcast(myname,msg)
            log@clear(logrec.logkey)

    on abort
        raiseException(fatalcomm_error)
end
```

no_agreement

No_agreement is called by the stage 3 exception handler of the *insert* procedure. *No_agreement* operates by first writing a "precommit bad" message to the log. This message indicates that the the *insert* operation should be rolled back, but for some reason th roll back was unsuccsessful. The roll back is retried at some later time by the *abort_pre* procedure in the recover object (section 10.2).

No_agreement handles disk and transmission exceptions as fatal errors. All other exceptions cause *no_agreement* to roll back the *insert* operation. The roll back is implemented by broadcasting an "abort insert action" and waiting for a reply. All receiving nodes invoke *ab_ins_act* which clears its local log and transmits an acknowledgement.

code

```
procedure no_agreement(logkey :key_type; except : exception_type) begin
    log@commit_type(logkey,"precommit bad")
    CASE exception OF
        fatal_disk_error : raiseException(fatal_disk_error)
        fatal_xmit_error : raiseException(fatal_xmit_error)
        mcast_unavailable : raiseException(mcast_unavailable)
        no_agreement :
            begin
                msg.type := "abort insert action"
                communicate@synch_bcast(myname,msg)
                communicate@synch_recv_all_yes(myname,msg)
                if no exception then log@clear(logkey)
                else raiseException(fatal_recovery_error(logkey)
                    no_exception : !do nothing
            end
    end CASE
    on abort raiseException(fatal_recovery_error(logkey))
end no_agreement
```

no_commit

The *no_commit* procedure is called by the stage 5 exception handler of *insert*. *No_commit* writes the message "bad commit" to the log and exits. Any exception is a fatal exception. The "bad commit" message indicates that the local node has proceded by committing a calendar entry, but some remote node may not have been notified. The recovery object will retry all remote nodes at some later time in the *recov_badcommit* (section 10.2) procedure.

code

```
procedure no_commit(key : key_type) begin
begin action nocom
```

```
    log@commit_type(key,"bad commit")

    on abort raiseException(fatal_commit_error)
end action nocom
end
```

receive

Receive is the central processing receive handler used by the calendar object. *Receive* invokes the correct code segment depending upon a received message's type.

code

```
procedure receive !blocking receive used by high level server process
begin
    communicate@synch_recv_any(src,msg)
    CASE msg.type OF
        "abort insert precommit" : ab_ins_pre(src,msg)
        "abort insert action" : ab_ins_act(src,msg)
        "insert precommit" : rec_insert_precommit(src,msg)
        "insert commit" : rec_insert_commit(src,msg)
        "status commit" : rec_stat_com(src,msg)
        "delete" : !not implemented
end
```

rec_insert_precommit

rec_insert_precommit is invoked when the node receives a "precommit" message. If the receiving node cannot insert the data into the calendar, the node returns "no", otherwise the node returns "yes" and commits the received entry to the log.

code

```
procedure rec_insert_precommit(src : IN name; msg : IN msg_type)
begin
begin action rec_precom
    stat := oktorecv(msg) !oktorecv not documented
        if (stat = TRUE) then begin
        msg.type := "received precommit"
        msg.data := "yes"
        log@commit(msg)
        end
    else msg.data := "no"
        IO@xmit(myname,src,msg)

        on abort
        CASE exception OF
            fatal_disk_error : raiseException(fatal_disk_error(msg.key))
            fatal_xmit_error : raiseException(fatal_xmit_error(msg.key))
        end CASE
end action
end procedure
```

rec_insert_commit

Rec_insert_commit is invoked when an "insert commit" message is received. The "insert commit" message indicates the completion of the second phase of the commit protocol. Insert commit writes the received message to the calendar, returns and acknowledgement, and clears the log record. Clearing the log record relinquishes the recovery object from querying for the status of the calendar entry in the event that the node recovers from a crash (see section 10.2).

code

```
procedure rec_insert_commit(src : IN name; msg : IN msg_type)
begin data area rec_in
    logrec : log_rec_type
end data area rec_in
begin
begin action rec_ins_com
    log@read(msg.key,logmsg)
    write_cal(msg)
    log@compute_key(logrec)
    msg.data := "yes"
    IO@xmit(myname,src,msg)
    log@clear(logrec)

    on abort
    CASE exception OF
        fatal_disk_error : raiseException(fatal_disk_error(msg.key))
        fatal_xmit_error : raiseException(fatal_xmit_error(msg.key))
    end CASE
end action
end procedure
```

ab_ins_pre

ab_ins_pre is invoked when an abort precommit message is received (see section 10.2). This procedure clears the log entry if the entry exists. If the entry does not exist, the read operation returns an exception that is not an error.

code

```
procedured ab_ins_pre(src : pending; msg : msg_type)
begin
```

```
begin action ab_ins_pre
    stage 1: log@read(msg.key,msg)
    stage 2: log@clear(msg.key)

    on abort
    CASE exception OF
        stage 1: !no error: do nothing precommit msg never received
        stage 2: raiseException(fatal_badlog(fatal_disk_error,msg.key))
end action
end procedure
```

ab_ins_act

ab_ins_pre is called whenever an "abort insert action" message is received. The "abort insert action" message is sent in the *no_agreement* procedure (section 10.2) which is called by the exception handler of stage 3 of the exception handler of *insert* (section 10.2).

code

```
procedure ab_ins_act(src : IN name, msg : msg_type)
begin data area ab_in
    logrec : log_rec_typ
end data area ab_in

begin
begin action ab_ins_act
    stage 1:
        log@read(msg.key,logrec)
        msg.data := "yes"
    stage 2: IO@xmit(myname,src,msg)
        log@clear(date,time,msg)

    on abort
```

```
    CASE stage OF
        stage 1: raiseException(abort_incompete(fatal_disk_error))
        stage 2: CASE exception OF
            fatal_disk_error : raiseException(abort_incomplete(fatal_disk_error))
            transmit_error : raiseException(abort_incomplete(transmit_error))
    end CASE
end action
end procedure
```

rec_stat_com

rec_stat_com is invoked whenever a "status commit" message is received. The status commit message is called by the recovery object (see section 10.2) when a log entry marked "bad commit" is encountered. A "bad commit" entry is inserted into the log by the *no_commit* procedure (see section 10.2) whenever the *insert* procedure is unable to guarantee consensus among the nodes of a committed entry.

code

```
procedure rec_stat_com(src : IN name; msg : IN msg_type)
begin
begin action rec_st_com
    stage 1: log@read(log@compute_key(date,time),msg)
    stage 2: msg.data := "yes"
             IO@xmit(calendar@myname,src,msg)
    stage 3: log@clear(log@compute_key(date,time))

    on abort
    CASE stage OF
    stage 1: if (exception = key_not_found) then begin
        msg.data := "yes"
        IO@xmit(calendar@myname,src,msg)
        log@clear(date,time,msg)
```

```
        if not (exception = key_not_found)
            raiseException(fatal_receive_error)
    stage 2: raiseException(fatal_receive_error)
    stage 3: raiseException(fatal_disk_error)
end action rec_st_com
end
```

queryIO

queryIO prompts the user for a calendar entry to be queried.

code

```
procedure queryIO(msg : OUT msg_type)
begin
    IO@write("date: ")
    IO@read(logrec.logmsg.date)
    IO@write("time: ")
    IO@read(logrec.logmsg.time)
    key = log@computekey(logmsg)
end
```

query

Query prompts the user for an entry to be queried. If a log entry exists and the log entry has the value "precommit" (see sections 10.2 and 10.2), then *query* returns the value stored in the log, otherwise, *query* returns the value stored in the calendar.

code

```
procedure query
```

```
begin
begin data area query_dat
    logrec : log_rec_type
end data area query_dat

    queryIO(date,time,key)
    if (log@exists_rec(key)) then begin
        log@read_log_rec(key,data)
        CASE data.type OF
            "precommit" : IO@write(logrec.logmsg.data)
            otherwise : begin
                calendar@read(key,logrec.logmsg)
                IO@write(logrec.logmsg.data)
            end
        end CASE
    else !no log record !
        calendar@read(key,logrec.logmsg)
        IO@write(logrec.logmsg.data)
end !procedure query
```

others

The other procedures are *lookup, write_cal, read, compute_key*, and *read_logrec* (implementation details omitted).

code

```
procedure lookup(key : IN key_type)returns msg_type
!Given a key, return the msg_type from the calendar
!If no such key is available the procedure aborts and raises the
!exception: 'msg_key_unavailable'
!
```

```
procedure write_cal(msg : IN msg_type)
!Force an entry into the calendar stored in stable storage. The entry
!can be looked up using the unique key.
!

procedure read(key : IN key_type) returns msg_type
!Given a key, return the corresponding value from the calendar.
!Raise exception: key_unavailable if the key cannot be found in the calendar
!

procedure compute_key(msg : IN msg_type) returns key_type
!Given msg.date and msg.time compute the unique key that names a calendar
!entry. We assume no two entries have the same node/date/time stamp.
!RaiseException: key_not_computed if an exception occurs
!

procedure read_logrec(key : IN key_type; logrec : OUT log_rec_type)
!Read the entry named by key from the calendar stored in stable storage (using
!the read operation) into a log record data structure
!


procedure myname returns name
!Return the unique name of the local node.
!
```

recovery object

The object recovery is invoked whenever a node recovers from a crash.

recover

recover rolls back the log. For each log entry, *recover* checks the *type* and dispatches to the appropriate procedure. *Recover* is called when the node recovers from a crash.

code

```
procedure recover
begin data area rec
   logrec : log_rec_type
end data area rec

   for each logrec := log@read do begin
   CASE logrec.logtype OF
      "insert precommit" : recover_precom(logrec)
      "precommit good" : recover_ins_com(logrec)
      "received precommit" : recover_rec_precom(logrec)
      "precommit bad" : abort_pre(logrec)
      "bad commit" : recov_badcommit(logrec)
   end CASE
   end for
end procedure
```

recover_precom

recover_precom is called when a node reaches stage 2 of *insert* (see section 10.2), writes the "insert precommit" message to the log, and then crashes. This procedure implements backward recovery. If a transaction is aborted during precommit stage, the transaction is simply aborted. An abort is implemented by sending an "abort insert action" message. The receiver invokes *ab_ins_act* (section 10.2) when the abort message is received.

code

```
procedure recover_precom(logrec)
begin data area ins_precom
    msg : msg_type
    name : pending
end data area ins_precom

begin
begin action rec_ins_pre
    logrec.logmsg.type := "abort insert action"
    communicate@synch_bcast(calendar@myname,logrec.logmsg)
    communicate@synch_recv_all_yes(myname,logrec.logmsg)
    log@clear(logrec.logkey)

    on abort raiseException(fatal_recovery_error(logrec.logkey))
    !note: on an exception the log is NOT cleared
end action
end procedure
```

recover_ins_com

Recover_ins_com is called if the insert procedure reaches stage 4 (see section 10.2) and then crashes. This procedure implements forward recovery by broadcasting a commit message.

code

```
procedure recover_ins_com(logrec)

begin
begin action rec_com
    stage 1:
        msg.type := "insert commit"
```

```
        communicate@synch_bcast(calendar@myname,"commit")
    stage 2:
        communicate@synch_recv_all_yes(calendar@myname,msg_array)
    stage 3:
        log@clear(logrec.logseq)

    on abort
    CASE stage OF
    stage 1 : raiseException(commit_comm_error)
    stage 2 : raiseException(commit_comm_error)
    stage 3 : raiseException(fatal_disk_error)
end action
end procedure
```

recover_rec_precom

Recover_rec_precom is called when the node receives a precommit message and then crashes. The node recovers by sending a *query* message message to see if the commit proceeded. This portion of the protocol is not included in this example.

abort_pre

abort_pre is invoked if the node crashes in the exception handlers of either stage 2 stage 3 of *insert* (see sections 10.2 and 10.2). The exception handlers call *no_precommit* (section 10.2) and *no_agreement* (section 10.2). *Abort_pre* is invoked only if the node crashes in *no_precommit* or *no_agreement.*

code

```
procedure abort_pre(logrec : log_rec_type)
begin
begin action ab_pre
```

```
    stage 1: communicate@synch_bcast(calendar@myname,msg)
    stage 2: log@clear(logrec.logkey)

    on abort
    CASE exception OF
        stage 1 : raiseException(fatal_recovery_error(bcast_unavailable))
        stage 2 : raiseException(fatal_recovery_error(disk_error))
    end !CASE
end !action ab_pre
end !procedure
```

recov_badcommit

Recov_badcommit is called if a "bad commit" message was placed in the log by *no_commit* (section 10.2). This message indicates a commit is unsuccessful even though all nodes agreed to commit the message. The *recov_badcommit* procedure retries the commit.

code

```
procedure recov_badcommit(logrec : log_rec_type)
begin
begin action ab_pre
    stage 1: logrec.logmsg.type := "status commit"
        communicate@synch_bcast(calendar@myname,logrec.logmsg)
    stage 2: communicate@synch_recv_all_yes(calendar@myname)
    stage 3: log@clear(logrec.logkey)

    on abort
    CASE stage OF
    stage 1: raiseException(fatal_recovery_error)
    stage 2: raiseException(no_consensus)
    stage 3: raiseException(fatal_disk_error)
    end !case
```

```
end action ab_pre
end
```

log object

The following procedures are access the log. *write, read, log_commit, commit_type, compute_key* and *clear*. The procedures are either self explanatory or documented below.

```
log_rec_type (export)
    logmsg : msg_type !import msg_type from the calendar object
    logkey : key_type !the unique key of the logrecord
    logtype : log_type_type !the type of the log record
    end
```

code

```
!log is the logging object.
implementation of object log_object

write(logmsg : IN log_rec_type)
!Force a log message and out to stable storage

read(key : IN key_type; logrec : OUT log_rec_type)
!Read the log record indicated by key into logrec
!

log_commit(logval : INOUT log_rec_type)
begin

begin action putlog
    log@write(logval)
```

```
    on abort raiseException(fatal_log_commit_error)
end action putlog
end logcommit

procedure commit_type(logkey : IN key_type; ltype : log_type_type)
!Update the type field of a log entry named by logkey to the value ltype
!on abort raiseException(fatal_log_commit m_error)
!

procedure compute_key(logval : INOUT log_rec_type)
!Given the date and time compute the unique key for a log record
!and place the key in the logval record
!

procedure clear(key : IN key_type)
begin
begin action logclear
    ! remove the entry named by key from the log
    on abort raiseException(key_not_found)
end action logclear
end object log
```

communicate object

Implementation of object communicate

```
!multicast a message to a group
!Irreversable action
procedure synch_mcast(IN src : name; dst : IN group_name; msg : msg_type)
                exceptions(port_disabled)
```

```
begin action m_cast
    begin data area 1
    src_port : capability
    dst_port_lst : array of capabilities
    end

    stage 1 : src_port := port@obtain_port(src)
        dst_port_lst := port@obtain_group_port(dst)

    stage 2 : for i := list@first(dst_port_lst) TO list@last(dst_port_lst)
            IO@xmit(src_port,port@translat(dst_port_lst,i),frame)
        end
    on abort

    case staged abort of
    stage 1 : raiseException(port_unavailable)
    stage 2 : raiseException(multicast_incomplete)
end


!Broadcast a message to every node on the network. Broadcast is
!implemented through a multicast sub action.
procedure synch_bcast(src : IN name;msg:msg_type); exceptions(port_disabled)

    synch_mcast(src,port@obtain_bcast_name,msg)
end


procedure synch_recv_any(src : IN name; msg : IN msg_type)
begin
begin action
    IO@recv_any(port@obtain_port(src),msg))
    on abort raiseExeption(comm_unavailable)
end action
```

```
end

!Receive a message from every port in dst_port_lst
procedure synch_recv(src : IN name;
        dst : dst_port_lst;
        msg : msg_type)

begin
begin action
    for i := list@first(dst) TO list@last(dst)
        parbegin
            IO@recv(src_port,port@translat(dst_port_lst,i),msg)
        parend
end action

on abort
    raiseException(fatal_recv_error)
end


!Receive a message from every node. This procedure blocks until every
!node sends a message.
!
procedure synch_recv_all(src : IN name
            msg_arr : OUT frame)

begin
    synch_recv(src,port@obtain_all_port,msg_arr)
end


procedure msg_validate(msg_arr :IN array of msg_type;
                val : IN string) returns(boolean)
!Return true if and only if every entry in the array msg_arr
```

```
!has the value "val"

!Receive a message from every node. Return success if and only if
!every node returns success
!
procedure synch_recv_all_yes(src : IN name)
begin data area yes
    msg_arr : array of msg_type
end data area yes
begin
    synch_recv_all(src,msg_arr)
    if not msg_validate@ACK_check(msg_arr,"yes")
        then raiseException(no_agreement)
end
end object
```

shell object

```
!Shell is the user interface object.
!
implementation of object shell !()
begin data area IO
    myname : name = pending ! my network name !
end data area IO

!User is the user interace. User is implemented as an infinite loop.
!The user may invoke a procedure (insert, query, or remove) by calling
!the appropriate procedure.
procedure user
begin data area 1
    token
    done = false
```

```
end data area 1

begin
begin action user
    REPEAT
    IO@prompt
    IO@get_token(token)
    CASE token OF
        insert : calendar@insert
        query : calendar@query
        remove : calendar@remove
        halt : done := true
    end
    UNTIL done
    on abort
    !Interact with user to recover from fatal errors
end action
end


!Return the local name of the host.
procedure local_name returns name
begin
    return(myname);
end;
```

References

[1] Andler, S., Ding I., Eswaran, K., Hauser, C., Kim, W., Mehl, J., and Williams, R., "System D", *Proceedings of the* 8^{th} *International Conference on Very Large Data Bases* Mexico City, Mexico (Sept 1982) pp. 33-44.

[2] Balchen, J., Mumme, K., *Process Control Structures and Applications*, (New York: Van Nostrand Reinhold Co., 1988).

[3] Bartlett, J. "A nonstop operating system" *Proceedings of the 1978 International Conference on System Sciences* (Honolulu, Hawaii, Jan. 1978).

[4] Bernstein, P., Hadzilacos, V., and Goodman, N., *Concurrency Control and Recovery in Database Systems*, (Reading MA: Addison-Wesley, 1987).

[5] Bhargava, B., *Concurrency Control and Reliability in Distributed Systems*, (New York: Van Nostrand Reinhold Co., 1987).

[6] Brownbridge, D.,L. Marshall, and B. Randell, "The Newcastle Connection, or UNIXes of the World Unite!" *Software Practice and Experience*, Vol. 12, Wiley Inter-science, Dec. 1982, pp. 1147-1162.

[7] Dasgupta P., LeBlanc R., Spafford E., "The Clouds Project: Design and Implementation of a Fault-Tolerant Distributed Operating System," Georgia Institute of Technology Technical Report GIT-ICS-85/29.

[8] Gehani, N., McGettrick, A., *Concurrent Programming*, (Reading MA: Addison-Wesley, 1988).

[9] Goodenough, J., "Exception Handling Design Issues," *ACM SIGPLAN Notices*, July 1975, pp. 41-45.

[10] Griffeth, N., Unpublished course notes for Database Design, Georgia Institute of Technology.

[11] Hwang, K., and Briggs, F., *Computer Architectures and Parallel Processing*, (New York: McGraw-Hill, 1984).

[12] Jovic, F., *Process Control Systems*, (Houston: Gulf Pub. Co., 1986).

[13] Kastner, P.C., "A fault- tolerant transaction processing environment", IEEE Q. Bull. Database Eng. 6, 2 (June 1983), special issue on Highly Available Systems.

[14] Kim, W, "Highly Available Systems for Database Applications", *ACM Computing Surveys*, Vol. 16, No. 1, March 1984, pp.71-98.

[15] Korth, H., Silberschatz, A, *Database System Concepts*, (New York: McGraw-Hill Book Co. 1986).

[16] Lamport, L., Shostak, R., and Pease, M., "The Byzantine Generals Problem," Technical Report 54, Computer Science Laboratory, SRI International, March 1980.

[17] Lin, T., and Siewiorek, D., "On-Line Fault Prediction in Distributed Environments: A Case Study" Carnegie-Mellon University.

[18] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C., "Abstraction Mechanisms in CLU", *Communications of the ACM*, Aug. 1977, pp. 564-576.

[19] Moss, J. Eliot B., *Nested Transactions: An Approach to Reliable Distributed Computing*, (Cambridge, MA: MIT Press, 1985).

[20] Stallings, W., *Data and Computer Communications*, (New York: Macmillan Publishing Co., 1988).

[21] *Stratus/32 System Overview*, (Natick MA: Stratus Computers)

[22] Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications*, (Englewood Cliffs, NJ: Prentice-Hall, Inc., 1982).

[23] Wright, Paul Kenneth and David Alan Bourne, *Manufacturing Intelligence*, (Reading, Massachusetts: Addison-Wesley, Inc., 1988).

Contents

1 Introduction

Among the benefits claimed for distributed computing are improvements in system fault tolerance and reliability, and increased availability of data and services. The *Clouds* project at Georgia Tech is one of a number of recent proposals in which reliability in a distributed system is based on the use of *atomic actions*, a generalization of the transaction concept of distributed databases. As part of the Clouds project, we have designed and implemented a high-level language providing access to the synchronization and recovery features of the Clouds system; this language is being used to implement those levels of the Clouds system above the kernel level. It also provides a framework within which to study programming methodologies suitable for systems based on the action concept, such as Clouds. Among the properties needed by systems data structures, the design of which must be addressed by such methodologies, are *resilience*—survivability and consistency of the data despite crashes and other faults; and *availability*—increased possibility of access to data despite network partitions or failures of some sites in a multicomputer system. Together with a mechanism that ensures *forward progress*—continued execution of jobs despite failures, these properties provide *fault tolerance* in the system.

In this paper, we describe some of the results of a study of methods of achieving fault tolerance in the Clouds system, in particular achieving resilience and increased availability of objects in Clouds. The remainder of this introduction presents the problems explored by this work. Section 1.3 describes the model of distributed computation in which the problems posed by the research were examined (the Clouds system) and the tools which were used to address these problems (the Aeolus[1] programming language). In Section 2, we describe various methodologies for achieving resilience using the tools provided by Aeolus/Clouds as well as discuss methods other researchers have proposed. In Section 3, we explore the various methodologies (both from Aelous/Clouds and others) proposed to achieve availability. The language runtime support features (primitives) required to support resilience as well as operating system support needed to support these features, are presented in Section 4. Finally, the language runtime support features and operating system support needed to support availability are discussed in Section 5.

[1] Aeolus was the king of the winds in Greek mythology.

1.1 The Need for Resilience

The distribution of a computation is of little benefit if the failure of a single site or portion of the network may leave that computation in an inconsistent state, or corrupt or destroy the data being operated upon by the computation. As noted above, distribution of physical resources alone may actually increase the chances of hardware failure, and thus the possibility of inconsistency. Thus, there is a need for mechanisms which ensure that failures in a distributed system do not leave the data at failed sites in a corrupted state, and that no inconsistencies are introduced in the data at operational sites because of computations which had visited the failed (or inaccessible) sites.

In the Clouds prototype, the resilience of data—that is, its consistency despite failures—is provided by the combination of *stable storage* and *action* mechanisms. Stable storage helps ensure that data is not corrupted by a failure at its site; the use of an action to provide a "firewall" around a computation helps ensure, through interaction with the stable storage facility, that no data at any site visited by that computation are left in an inconsistent state. The organization of data into *objects*—that is, containers for data which allow access to that data only through *operations* which they provide—simplify the tasks of the stable storage and action mechanisms by delimiting the effects of changes (and thus of failures), and by providing a definition of the state of a distributed computation. These concepts are explained in more detail in Section 1.3.

1.2 The Need for Availability

Even if a computation is distributed, it is subject to a single point of failure if any of the data objects involved in that computation exist at only a single node. The provision of resilience alone cannot eliminate the problems caused by site or network failures; although inconsistencies introduced by such failures have been abolished, any objects existing only at a failed site are unavailable for the duration of the failure, and thus no computation may proceed which requires those objects. A method for eliminating these bottlenecks is *data replication*, that is, the maintenance of copies of an object at multiple sites.

The use of replication introduces the problem of maintaining the *consistency* of the individual replicas when operations are executed on them. A common requirement for consistency is that the replicated object maintain *single-copy semantics*, that is, that the

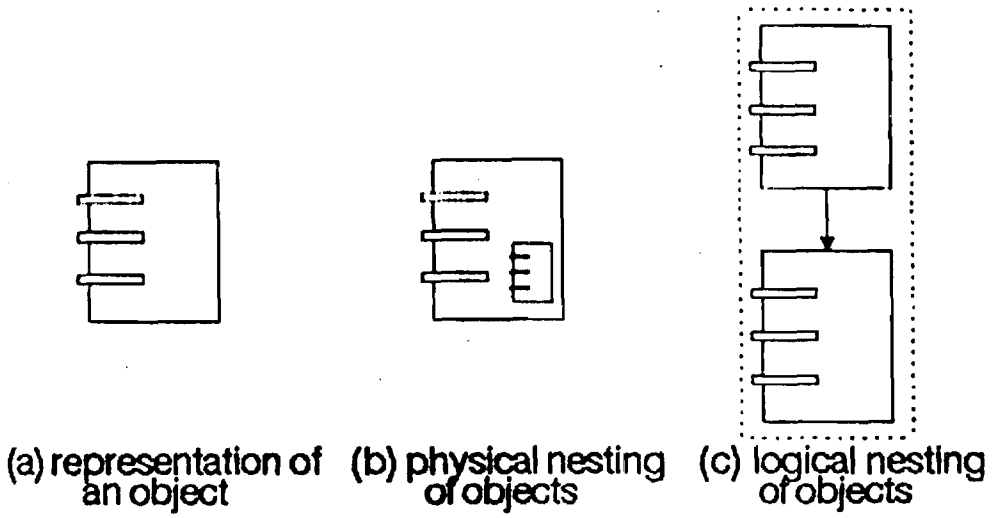

Figure 1: Pictoral Representation of Object Nesting

state of each replica be consistent with that which would have been obtained had the object existed only at a single site and had the same sequence of operations been applied to it. This is achieved by a combination of a mechanism for controlling concurrency among the replicas, and of a mechanism for copying the state obtained by an operation execution among the replicas.

These mechanisms have been the subject of much study, both in the areas of database systems and of operating systems. Indeed, it has been found that single-copy semantics is too stringent a requirement in some applications. (See [Wilk87] for a discussion of previous work in this area.) However, most previous work on such mechanisms has been concerned with "flat" data, such as files. The unique problems posed for these mechanisms by the object construct used in systems such as Clouds are discussed in the following section; in so doing, we also introduce some terminology used in the remainder of this paper.

1.2.1 Problems of Replication in Object-Based Systems

In the course of research on methods of achieving availability in object-based systems such as Clouds, we have found that the generality of the abstract object structure supported by Clouds poses problems for replication methods which are not presented by a less general, flat object structure (for instance, files or queues).

The problem lies in the possibility of the arbitrarily complex *logical* nesting of Clouds

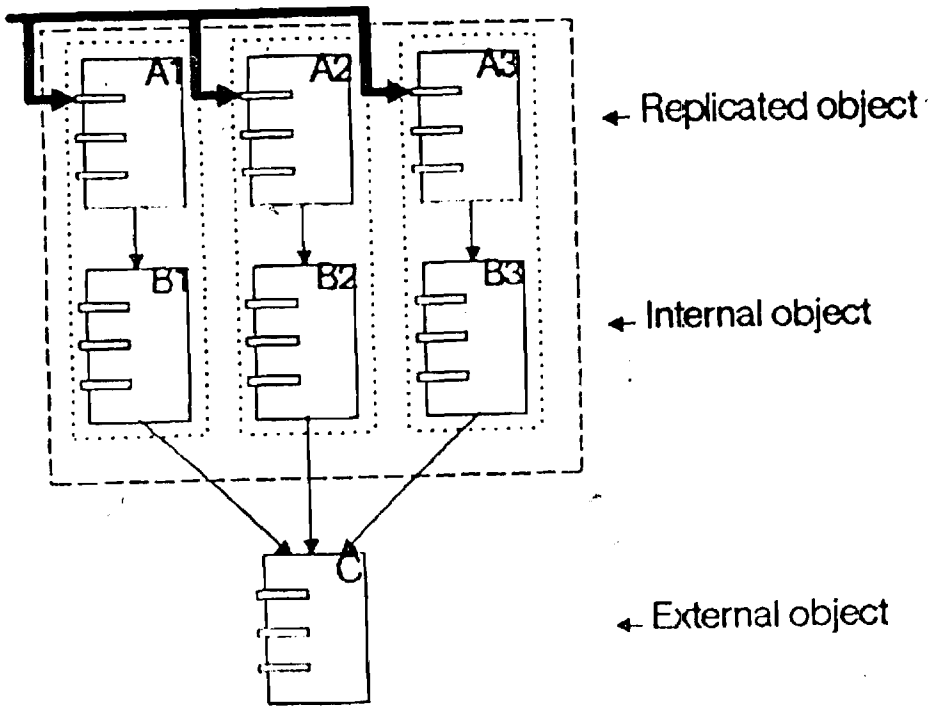

Figure 2: Replicated Object with Internal and External Object References

objects. Although Clouds objects may not be *physically* nested (that is, one object may not physically contain another object), an object may contain a capability to another object. If an object A creates another object B, and retains sole access to B's capability (by refraining from passing the capability to other objects, either explicitly or through an intermediary such as an object directory service), object B is said to be *internal to* object A. The internal object B may be regarded as being *logically* nested in object A. (A pictorial representation of physical and logical nesting is shown in Figure 1.) If, on the other hand, object A passes B's capability to some object not internal to A, or if A registers B's capability with an object directory service, B is said to be an *external* object; an external object is potentially accessible by objects not internal to the object which created the external object.

Problems arise with replication schemes when internal and external objects are mixed together in the same structure, *i.e.*, when an object may contain capabilities to both internal and external objects. (An example of such an object is represented in Figure 2.) These problems are associated with the method which is used to propagate the state of a replicated object among its replicas. One such method is to execute **at each replica** the computation from which the desired state results; this scheme is called *idemexecution*. Another method

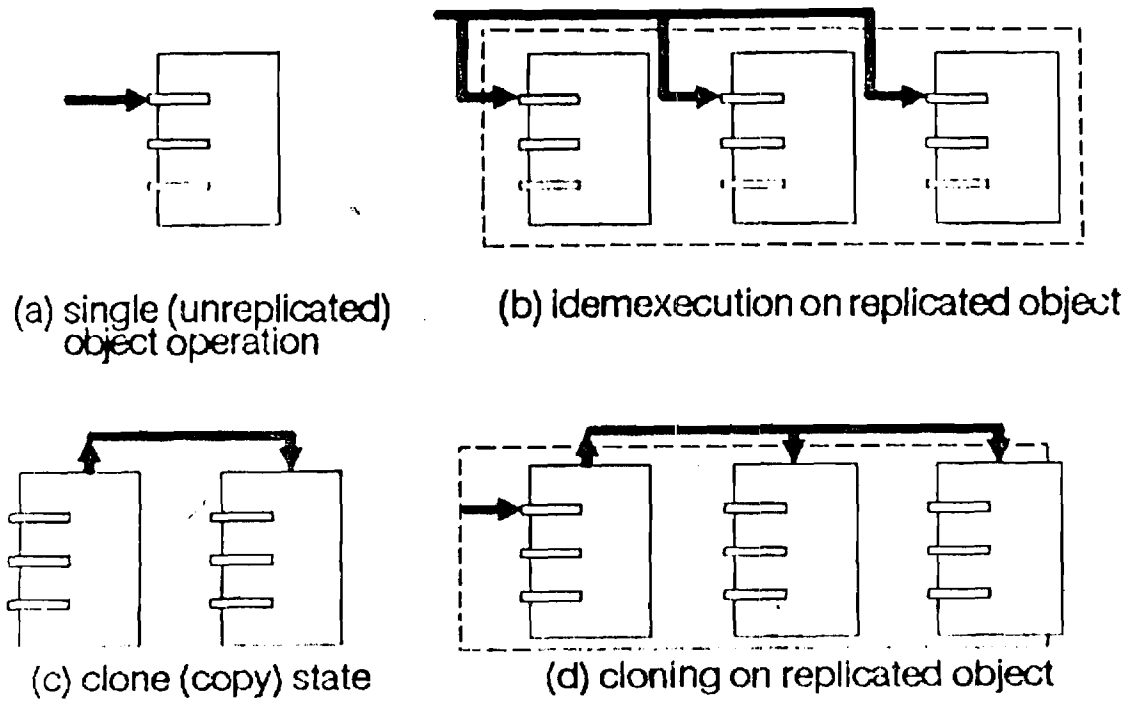

Figure 3: Replicated State-Copying Methods

is to execute the computation at one replica, and then copy the state of that replica to the other replicas; this scheme is called *cloning*. (Representations of the idemexecution and the cloning methods are shown in Figure 3.) Note that the scheme which is used to ensure that the replicas maintain consistent states (*e.g.*, quorum consensus) is not involved in these problems, and is considered separately in this investigation.

External objects cause problems when idemexecution is used to propagate state among replicas. If the replicated object performs some operation on an external object (*e.g.*, a print queue server), then—under idemexecution—that operation will be repeated by each replica. If the operation being performed on the external object is not idempotent, this can cause serious problems (*e.g.*, multiple submissions of a job to the print queue). Also, trouble may arise due to idemexecution if the operation on the external object is non-deterministic (for instance, random number generation, or disk block allocation among multiple concurrent processes).

On the other hand, internal objects cause problems when cloning is used to propagate state. For example, assume that each replica of an object creates a set of internal objects. Then, when an operation is performed on one of the replicas, its state—under cloning—is

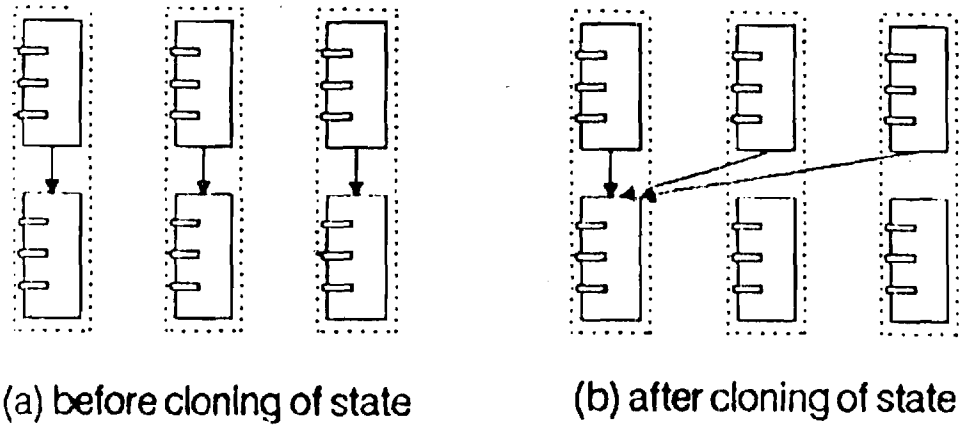

Figure 4: State Cloning with Internal Objects

copied to each of the other replicas. However, the capabilities to the internal objects of the replicas are contained in their states; thus, each replica now contains capabilities to the internal objects of that replica on which the operation was actually performed, and the information about the internal objects of the other replicas is lost. This problem is illustrated in Figure 4. In Figure 4 (a), each replica has a capability to its individual internal object. In Figure 4 (b), an operation execution has taken place at the leftmost replica in the figure, and its state has been cloned to the other two replicas; the states of the other replicas now contain capabilities to the internal object of the leftmost replica rather than to their own internal objects.

1.3 The Aeolus/Clouds Model

In this section, we provide an overview of the model of distributed computation embodied in the Aeolus/Clouds system. The background of the Clouds distributed operating system project, as well as the major concepts and facilities presented by the Clouds system, are presented here; a more complete description of the system may be found in a recent overview paper. [Dasg87] Also, the major features of the *Aeolus* language are described briefly.

1.3.1 The Clouds System

The Clouds distributed operating system project has been under development at Georgia Tech since late 1981; the central concepts were developed by Allchin and McKendry in a pair of early papers, [Allc82] [Allc83] and the Clouds architecture was described in full in

Allchin's dissertation. [Allc83a] The goal of the Clouds project is the implementation of a fault-tolerant distributed operating system based on the notions of *objects, actions,* and *processes,* to provide an environment for the construction of reliable applications on unreliable hardware. The basic approach is to exploit the redundancy available in distributed systems which consist of multiple computers connected by high-speed local area networks. Such systems are called *multicomputers* or *computer clusters.* In Clouds, the notion of an *object* may be used to represent system components, such as directories or queues. A set of changes to objects may be grouped into an *action,* which corresponds roughly to the *transaction* concept of distributed database work, providing an "all or nothing" assurance of atomic execution (a property sometimes called *failure atomicity*). The underlying support system ensures that, even if the actions extend across multiple machines, the changes will occur in totality or not at all. At this level, the support system, known as the *Clouds kernel,* is maintaining the *consistency* of the objects. It ensures that objects either reflect the effects of an action totally or not at all—no intermediate states are possible. This guarantee of an action's totality permits one to characterize the effects of hardware component failures: they cause actions to fail. Since a failed action is guaranteed to have had no effects on the objects with which it interacted, the action may be restarted without concern for potential inconsistencies it might have created.

Actions in Clouds go beyond the related notion of transactions in a database system. Rather than modelling all access to objects as simple reads or writes, the Clouds approach supports arbitrary operations on objects and allows a programmer to take advantage of operation semantics to increase concurrency, and thereby, performance. Through appropriate use of encapsulation, concurrent actions can be allowed to change objects without violating serializability.

A powerful feature of Clouds is the separation of the two components the traditional notion of the serializability of atomic actions, *failure atomicity* and *view atomicity.* Failure atomicity, as mentioned above, refers to the "all or nothing" property of atomic actions; view atomicity requires that the effects of an uncommitted action are not seen by other actions until commital occurs, thus avoiding the problem of "cascading aborts" of actions which have viewed intermediate states of an uncommitted action that later is aborted. This separation of the recovery and synchronization aspects of serializability allows the Clouds programmer to design objects that, while maintaining an appearance of serializability to the outside world, may violate strict serializability internally—in ways based on the pro-

grammer's knowledge of the object's semantics—in the interest of system efficiency.

Objects, actions, and processes are fundamental concepts supported by the Clouds architecture. To support these concepts, recovery and consistency are incorporated into the basic virtual memory mechanism. [Pitt86] [Pitt87] Synchronization mechanisms to control the interactions of actions are also provided. It is with these capabilities that Clouds is meant to support the data integrity required for the implementation of reliable, distributed application programs.

The detailed design of the Clouds kernel is discussed in Spafford's dissertation. [Spaf86] A prototype of the Clouds kernel, also described by Spafford, has been implemented on a hardware testbed consisting of VAX [2] 750s connected by a 10Mbps Ethernet, several dual-ported disk drives, and Sun 3 Workstations [3] running UNIX[4] —also attached to the Ethernet—that provide a user interface to the Clouds system. The Clouds kernel is implemented "on the bare machine," that is, it is not implemented on top of some other operating system such as UNIX. Thus, the features of objects, actions, and processes have been implemented in the lowest levels of the kernel, allowing use of the Clouds concepts in the construction of the operating system itself. At these lowest levels, we attempt to avoid implementing *policies*, instead providing *mechanisms* with which policies may be constructed. Some policies are embedded in subcomponents of the kernel. The storage management system [Pitt86] implements support for action-based stable storage within the object virtual memory mechanism. The action manager [Kenl86] controls the interaction of actions with objects, including creation, committal, and abortion of actions, a time-based orphan detection facility. and support for lock-based synchronization. Those kernel subcomponents implementing policy are intended to be replacable with minimal changes to the rest of the kernel. For instance, the storage management system could be replaced with another implementing log-based recovery, or the action manager changed to support timestamp-based synchronization, without fundamental changes to other kernel subcomponents.

The Clouds system above the kernel level consists of a set of fault-tolerant *servers* which provide system services (such as object filing, job scheduling, printer spooling, and the like) to application programs. (It is for the construction of this level of the Clouds system that the Aeolus programming language was designed; the kernel itself has been implemented in

[2]VAX is a registered trademark of Digital Equipment Corp.

[3]Sun Workstation is a registered trademark of Sun Microsystems, Inc.

[4]UNIX is a registered trademark of AT&T.

the C language.)

The location-transparency and resilience mechanisms provided by the Clouds architecture are used to support the operating system itself and its services. Thus, the system itself is decentralized (in the sense that the system can survive the failure of any node) and resilient. The Clouds system may be considered to consist of a set of fault-tolerant objects which in combination provide a reliable environment for applications.

1.3.2 The Aeolus Programming Language

In this section we provide a brief overview of the Aeolus programming language. More complete discussions of Aeolus may be found in previous publications. [Wilk85] [Wilk86] [Wilk87]

Aeolus developed from the need for an implementation language for those portions of the Clouds system above the kernel level. Aeolus has evolved with these purposes:

- to provide the power needed for systems programming without sacrificing readability or maintainability;
- to provide abstractions of the Clouds notions of objects, actions, and processes as features within the language;
- to provide access to the recoverability and synchronization features of the Clouds system; and
- to serve as a testbed for the study of programming methodologies for action-object systems such as Clouds. [LeBl85]

The intended users of Aeolus are systems programmers working on servers for the Clouds system. Clouds provides powerful features for the efficient support of resilient objects where the semantics of the objects are taken into account; it is assumed that the intended users have the necessary skills to make use of these features. Thus, although access to the automatic recovery and synchronization features of Clouds is available, we have avoided providing very-high-level features for programming resilient objects in the language, with the intention of evolving designs for such features out of experience with programming in Aeolus.

Aeolus provides access to the action manager's support construct. An unusual aspect of Aeolus/Clouds locks is tha the specific data being locked, but rather with values in so obtained for a value of an object, and not on the object itself. be obtained on a file name even if that file does not yet exis of Aeolus/Clouds locks is that they provide a mechanism fo locking *modes* and arbitrary compatibilities between the diff lock to be tailored to the specific synchronization semantics c For example:

```
type file_lock is lock ( read  : [ read ],  write
        domain is string( FILE_NAME_SIZE )
```

The declaration of `file_lock` defines a lock type over the filenames, in which the usual multiple reader/single writer the compatibilities among the `read` and `write` modes of th

All locks obtained during execution in the environment and propagated to the immediate ancestor of that action u released by the programmer. Locks obtained under an acti the action aborts or successfully performs a toplevel comm protocol (2PL) is maintained, with violations to 2PL allow if the programmer deems such violations acceptable. A lock a nested action even if conflicting locks are held under one action, but not if conflicting locks are held under an action nested action. [Allc83a] The power of the Aeolus/Clouds lo defined synchronization lies in the specification of arbitrar compatibilities between those modes, as well as the dissoc variables.

Support for Objects The *object* construct provides sup olus. A collection of related data items may be *encapsulat* may provide *operations* (procedures that operate) on the da of an object is via these operations; thus, an object can stri encapsulated data, helping guarantee the invariants of the

the object defines a type, called an *object type,* which may be used in the declaration of variables to hold capabilities to *instances* of that object type.

Aeolus provides a hierarchy of *object classifications* sharing a common implementation and invocation syntax which offers a trade-off of functionality and efficiency. The object classifications fall into two groups: the so-called Clouds object classifications (`autorecoverable`, `recoverable`, and `nonrecoverable`) may make use of the object management facilities and (for `autorecoverable` and `recoverable` types) the action management facilities, while the non-Clouds object classifications (`local` and `pseudo`) do not use any of the Clouds facilities for action or object management and provide data-abstraction facilities usable "locally" (without resorting to the system facilities supporting distribution of objects). On the other hand, the Clouds object classifications provide access to the support for data abstraction provided by the Clouds system when the expense of that support is warranted; the separate classifications of Clouds objects allow the programmer to specify the degree of support (and of incurred expense) required. The object classifications are described in more detail in the papers cited above; while the *autorecoverable* classification provides the paradigm most often presented by other action systems, that is, completely automatic recovery of the entire object state, the *recoverable* classification is of more interest here in that it allows the programmer to tailor object recovery based on the semantics of the object via mechanisms described below.

The global variables of an object are called collectively the object's *state.* In an object of class `recoverable`, part of the object state may be specified to be in a *recoverable area;* also, the programmer may specify an *action events part* and/or a *per-action variables part.* Recoverable areas, action events, and per-action variables are described in more detail in section 2.

In order to allow the object to participate in its own creation and deletion, an object implementation part contains specifications of handlers for the so-called *object events.* The object events include the *init* or object initialization event, the handler for which is executed whenever an instance of the object is created by use of an allocator; the *reinit* or object reinitialization event, the handler for which is executed—if the object has registered its desire for reinitialization with the action manager—when the system is reinitialized after a crash or network partition; and the *delete* or object deletion event, the handler for which is executed when the object instance is destroyed.

An invocation of an object operation looks much like a procedure invocation, except that, outside the implementation part of the object itself, an operation name must be qualified by the name of a variable representing an instance of that object type (or, for pseudo-objects, by the name of the object type itself). Thus, for an instance of a bounded-stack type, the programmer might write

```
stack_instance @ push( elem )
```

When an object invokes one of its own operations, however, the usual procedure call syntax is used.

Invocations of pseudo-object and local object operations have semantics essentially similar to those of calls to procedures local to a compiland. The situation is different for operations declared in objects which use the Clouds object-management facilities (*i.e.*, the so-called "Clouds objects"). Invocations of operations on Clouds objects are handled by the compiler through operations on the Clouds object manager on the machine on which the invoking code is running. The Clouds object on which the operation is being invoked need not be located on the same machine as the invoking code; the object manager then makes a *remote procedure call* (RPC) to the object manager on the machine on which the called object resides. The location—local or remote—of the object being operated upon, however, need not concern the programmer, as the RPC process is transparent above the object-management level.

Support for Actions The action concept provides an abstraction of the idea of work in the Clouds system; an action represents a unit of work. Actions provide *failure atomicity,* that is, they display "all-or-nothing" behavior: an action either runs to completion and *commits* its results, or, if some failure prevents completion, it *aborts* and its effects are cancelled as if the action had never executed.

Support for actions in the Aeolus language is relatively low-level. At present, the methodology of programming with actions is not as well-understood as the methodology of programming with objects; thus, rather than providing high-level syntactical abstractions such as those available for object programming, Aeolus allows access to the full power and detail of the Clouds system facilities for action management. The major syntactic support provided by Aeolus for action programming is in the programming of *action events, recoverable areas, permanent* and *per-action variables,* and *action invocations.*

At several points during the execution of an action, the action interacts with the *action manager* of the Clouds system to manage the states of objects touched by that action, including writing those states to *permanent* (stable or safe) storage, and recovering previous permanent states upon failure of an action. Thus, failure atomicity may be provided by the action management system. The *action events* include:

BOA beginning of action

toplevel_precommit prepare for commit of a toplevel action

nested_precommit prepare for commit of a nested action

commit normal end of action (EOA)

abort abnormal end of action

The interactions with the Clouds action manager necessary when such events take place are done by default procedures supplied by the Aeolus compiler and runtime system; these procedures are called *action event handlers.* When an action event occurs for a particular action, the action manager(s) involved invoke the event handlers for each object touched by that action.

The right-hand side of an assignment statement may take the form of an *action invocation.* Here, the right-hand side (which consists of an operation invocation which, if the operation is value-returning, is embedded in another assignment statement) is invoked as an action; the *action ID* of this action is assigned to the variable designated by the left-hand side of the action invocation. Thus, for example, if the bounded-stack object mentioned above were defined as a recoverable object, one might invoke one of its operations as an action:

```
aID := action( stack_instance @ push( elem ) )
```

The action ID may be used as a parameter in operations on the action manager which provide information about the status of the action, cause a process to wait on the completion of an action, or explicitly cause an action to commit or abort. By use of additional syntax not shown here, the programmer may specify that an action be created as a "top-level" action, that is, as an action with no ancestors; a top-level action cannot be affected by an

abort of any other action. Otherwise, the action is created as a "nested" action, that is, as a child (in the so-called *action tree*) of the action which created it; as described below, a nested action may be affected by an abort of one of its ancestors. Optionally, a *timeout value* may be specified in the action invocation clause; if the action has not committed by the expiration of this timeout, the action will be aborted. If no timeout value is specified, a system-defined default value is used. The detailed semantics of action invocations, and requirements on objects that may have operations invoked as actions, are described in the papers on Aeolus cited above.

2 Language Features for Resilience

2.1 Autorecoverable

As was described in Section 1.3, by use of the **autorecoverable** class of object, the programmer may take advantage of the recovery facilities of the Clouds system by having the compiler generate the necessary code automatically. This automatic recovery mechanism requires recovery of the entire state of the object, and uses the default action event handlers. However, it is sometimes possible for the programmer to improve the performance of object recovery by providing one or more object-specific event handlers which make use of the programmer's knowledge of the object's semantics; these programmer-supplied event handlers then replace the respective default event handlers for that object. Thus, if object class keyword **recoverable** is specified in the definition header of the object being implemented, the programmer may give an optional *action event part* in the object's implementation part. Following the keywords **action events**, the programmer lists the name of each action event handler provided by the object implementation as well as the name of the action event whose default handler the specified handler is to override. Thus, for example, the specification (in an object implementing a bounded-stack abstraction):

```
action events
   stack_BOA overrides BOA,
   stack_nested_precommit overrides nested_precommit
```

indicates that the default handlers for the **BOA** and **nested_precommit** action events are to be replaced by the procedures named **stack_BOA** and **stack_nested_precommit**, respectively, for the bounded-stack object type only.

2.2 Recoverable

As mentioned above, if an object being implemented is of class **recoverable**, then some of its variables may be declared in a **recoverable** area. When a nested action first invokes an operation on a recoverable object ("touches" that object), the action is given a new *version* of the recoverable area which initially has the same value as the version belonging to the action's immediate ancestor. The set of versions belonging to uncommitted actions

which have touched a recoverable object is maintained on a *version stack* by a Clouds action manager. When a nested action commits, its version replaces that of its immediate ancestor. When a toplevel action commits, its version is saved to permanent storage. If an action is aborted, its version is popped from the version stack. Thus, recoverable areas (in conjunction with appropriate use of synchronization) provide *view atomicity,* that is, an action does not see the intermediate (uncommitted) results of other actions. Also, the use of recoverable areas allows the programmer to provide finer granularity in the specification of that part of the object state which must be recoverable, since the use of automatic recovery on an object (the **autorecoverable** object class) requires recovery on the entire state of the object. The interaction with the action manager necessary to manage the states of recoverable areas is implemented by the action event handlers as described above. Again, the default event handlers may be overridden by programmer-supplied event handlers for the entire object to achieve better performance.

2.3 Per-action and permanent variables

It may sometimes be desirable to make large data structures resilient. In such cases, the recoverable area mechanism may be inefficient, since it requires the creation of a new version of the entire recoverable area for each action which modifies the area. Often in such cases the programmer may take advantage of knowledge of the semantics of the data structure to efficiently program the recovery of the data structure. The Aeolus language provides two constructs which aid in the custom programming of data recovery, the so-called *permanent* and *per-action variables,* constructs proposed by McKendry. [McKe85]

Any type may be given the attribute **permanent**. This attribute indicates that members of that type are to be allocated on the *permanent heap,* a dynamic storage area in the object storage of each object instance. This area receives special treatment by the Clouds storage manager; in particular, it is *shadow-paged* during the **toplevel_precommit** action event.

Aeolus also provides the per-action variable construct. A per-action variable specification resembles a recoverable area specification, and its semantics is also similar, in that each action which touches an object with per-action variables gets its own version of the variables; however, the programmer may access the per-action variables not only of the current action, but also of the parent of the current action. Also, per-action variables are allocated in *non-permanent storage,* that is, in storage the contents of which may be lost

upon node failure. The variables in a per-action specification are accessed as if they were fields in a record described by the specification; two entities of this "record type" are implicitly declared: `Self` and `Parent`, which refer respectively to the per-action variables of the current action and its immediate ancestor.

Permanent and per-action variables may be used together to simulate the effect of recoverable areas at a much lower cost in space per action. In general, the per-action variables are used to propagate changes to the resilient data structure up the action tree; these changes are then applied during the `toplevel_precommit` action event to the actual data structure in permanent storage. The use of permanent and per-action variables is shown more fully in the Aeolus papers cited above.

2.4 Resilient types

Experience with programming with the features controlling interaction of actions and objects has led to the development of a higher-level language construct to embody the observed methodology. This feature, called the *resilient type*, is described in this section.

A methodology for simulating the effects of recoverable areas without incurring the cost of multiple versions of the entire recoverable area was described above. This scheme involves the use of the **per-action** variable construct provided by Aeolus to maintain lists of the *intentions* of an action to modify the resilient state of an object. The resilient state is specified by use of the **permanent** variable construct of Aeolus. The lists of intended modifications are then used at top-level commit to perform the actual modifications to the resilient state.

Examples of the use of this methodology are provided in Appendices A and C. The design of such examples has led to the development of a declarative syntax to replace the imperative combination of the per-action/permanent variable constructs. This new construct is called the *resilient type.*

When using per-action and permanent variables to achieve resilience of permanent data, the programmer must specify the following characteristics:

- the representation of the permanent version of the data;
- the relationship of the `modifies` operations of the object to the permanent represen-

tation; and

- the visibility of both the permanent version and uncommitted modifications made to it by actions.

The first characteristic is achieved in the per-action/permanent variable paradigm by the specification of a permanent variable. The second characteristic is implemented by use of per-action variables to maintain lists of changes to the permanent variables made by each `modifies` operation; the programmer must specify in a top-level precommit action event handler how these modifications are to affect the permanent data. The third characteristic is realized typically by the use of a "lookup" function that takes into account both the permanent state and the uncommitted changes maintained in the per-action variables in some manner appropriate to the semantics of the object.

The use of the per-action and permanent variable constructs in this paradigm has two undesirable consequences: not only must the programmer explicitly specify exactly **how** the paradigm is to be implemented, but the implementation is scattered among many parts of the object, *i.e.*, the data and per-action variable declarations, `modifies` operations, and action event handlers. Thus, there is motivation to abstract the experience with the imperative constructs into the design of a higher-level, declarative feature that allows the programmer to specify **what** the characteristics of the resilient data are, rather than **how** these characteristics are to be achieved.

A preliminary design has been developed for a feature called the resilient type that expresses the three characteristics of the per-action/permanent variable paradigm in a declarative fashion. An example of a resilient object using this feature is presented in Appendix D. The declaration of the resilient type from this object is also shown in Figure 4. The syntax of the resilient type describes the characteristics of the type in the following order: representation of the permanent data; relationship of the `modifies` operations of the object to this data; and the visibility rule which applies to the permanent data and uncommitted modifications. The effect of a resilient type declaration is as follows. For each `modifies` operation, an "intentions list" is maintained by the system; at each invocation of such an operation, the values of those operation parameters which are specified in the `with modifies operations` clause are added to the list for that operation. The programmer may specify that one operation "reverses" the effect of another; in this case, the value inserted previously on the list of the reversed operation is removed. These lists are propagated to the parent

action upon nested commit. During the top-level precommit event, these lists are traversed using a "thunk" of code specified by the programmer in the **with modifies operations** clause. The visibility rule for a variable of the resilient type may be accessed by using the variable name as an object instance name, and invoking operation **visibility** on it. The representation of the permanent data structure may be accessed within the resilient type specification by the name **rep**. For instance, in the example of Figure 4, the entry for the **delete** operation of **symtab** is as follows:

```
delete ( name ) reverse insert :
   rep[ hash( name ) ] @ remove( name )
```

The effect of this entry is the maintenance of an intentions list for the **delete** operation. The programmer has specified that the **delete** operation reverses the effect of a previous **insert** operation by the same action. Thus, if an entry with value **name** is found on the intentions list for **insert**, that entry is removed from the list; otherwise, an entry is added to the intentions list for **delete**. During top-level precommit, the intentions list for **delete** is iterated using the specified thunk; its effect is to invoke the **remove** operation on the element (given by the **hash** function, which is not shown here) of the array of **bucket_list** objects that forms the permanent data structure of the resilient type (accessed by the name **rep**).

A final aspect of the resilient type specification bears explanation. It was found necessary to provide some way of accessing elements not only of the permanent data, but of the (visible) uncommitted results of **modifies** operations; such access is useful for displaying all visible elements of a resilient type, or for other operations requiring mapping-like functions. Thus, the final portion of the prototype syntax allows the programmer to specify an *iterator* function which can yield successive visible elements of the resilient type.

2.5 Other work

2.5.1 ISIS

The ISIS system developed at Cornell [Birm84] [Birm85] supports *k-resilient* objects (objects replicated at *k+1* sites and which can tolerate up to *k* failures) by means of checkpoints

```
type symtable_type is
   resilient array[ hash_range ] of bucket_list
   with modifies operations
      insert ( name, value ) :
         rep[ hash( name ) ] @ add( name, value ) ,
      delete ( name ) reverse insert :
         rep[ hash( name ) ] @ remove( name )
      end operations
   visibility ( name : name_type, out value : value_type ) is
         insert( name, value )
      or (     not delete( name )
           and rep[ hash( name ) ] @ find( name, value ) )
      end visibility
   iterator ( out value : value_type ) returns name_type is insert :
         for i in bucket_range loop
            return rep[ i ] @ iterate( value )
         end loop
      end iterator
   end resilient
```

Figure 5: Example of a Resilient Type Declaration

and the "available copies" algorithm. ISIS objects can refer to other objects, although apparently all such "nested" objects are considered to be external. This system provides both availability and *forward progress;* that is, even after up to k site failures, enough information is available (at the remaining sites possessing an object replica) that work started at the failed sites can continue at these remaining sites. This is accomplished through a *coordinator-cohort* scheme, where one replica acts as master during a transaction to coordinate updates at the other, "slave" replicas ("cohorts"). The choice of which replica acts as coordinator may differ from transaction to transaction. The object state is apparently copied from the coordinator to the cohorts via a cloning operation; this operation has been described as propagating a checkpoint of the entire coordinator, [Birm84] or, in a more recent paper, as propagating the most recent version in a version stack. [Birm85] In the current system, it is assumed that the network is not subject to partitioning.

In ISIS, a transaction is not aborted when a machine on which its coordinator is running fails (transactions are usually aborted only when a deadlock situation arises). Rather, the transaction is resumed at a cohort from the latest checkpoint, in what is called *restart mode;* this cohort becomes the new coordinator. Operations which the coordinator had executed after the latest checkpoint took place must be re-executed at the new coordinator.

In the course of an operation on a k-resilient object, the coordinator may perform operations on other objects to which it contains references. Such operations on "nested" objects are called *external actions.* Inconsistencies can arise due to external actions performed during restart mode; operations performed on external objects by the new coordinator in this mode were also performed by the old coordinator before it failed. Thus, unless the operations on external objects are idempotent, inconsistencies can arise. (This problem is closely related to the problem of idemexecution on external objects, discussed in Section 1.) This problem is solved in ISIS by requiring external objects to retain results of operations; these retained results are associated with a transaction ID. When a new coordinator takes over from a failed coordinator and enters restart mode, it uses the same IDs for its external operations, and rather than re-execute these operations, the external objects merely return the associated results.

There is also an idemexecution scheme due to Joseph [Jose85] [Jose86] which was apparently implemented as an experiment using the ISIS system as a testbed, rather than as part of the ISIS replication mechanism itself. In Joseph's scheme, the coordinator performs the requested operation, and then instructs its cohorts to perform the same operation.

Recently, a new version of the ISIS system, called ISIS-2, has been designed; it is anticipated that this new system will be operational by Fall 1987. The ISIS-2 design exploits a new abstraction called the *virtually synchronous process group.* [Birm87] In this abstraction, a distributed set of processes cooperate to perform work in an environment in which broadcasts, failures, and recoveries are made to appear synchronous.

2.5.2 Argus

The Argus system at MIT [Lisk83] [Lisk84] [Lisk83a] [Weih83] is a language and system for distributed applications which has evolved from the CLU language. Argus provides an object construct (called *Guardian*) which encapsulates data and processes, giving an abstraction of a physical node or server. Argus also retains the *cluster* construct from CLU, which provides functionality similar to that of local objects in Aeolus; however, the syntax of Guardians is not similar to that of clusters. Resilience in Argus is based on the notion of system-provided primitive *atomic data types*, from which user-defined atomic data types may be constructed. These primitive atomic data types also define the synchronization properties of the user-constructed types. Experience with programming a distributed, collaborative editing system in Argus has been described by Greif *et al.*; [Grei86] one criticism arising out of this experience was that they were sometimes forced to use a Guardian where a cluster might have been more appropriate.

An *atomic data type*, like a regular abstract data type, provides a set of objects and a set of operations which are the only method of interacting with the object. Unlike regular types, however, an atomic type provides serializability and recoverability for actions that use objects of that type. To achieve atomicity in Argus there were two requirements imposed on the execution sequences that affected objects. To support recoverability they required that actions can observe the effects of other actions only if those actions committed. This ensures that aborted actions cannot have any effect on other actions. The second requirement was needed to help insure serializability. Operations executed by one action cannot invalidate the results of operations executed by a concurrent action.

In Argus, an application is implemented from one or more modules called *guardians.* Each guardian consists of some data objects and some processes to manipulate those objects. Sharing of objects between guardians is not permitted. *Handlers,* a set of operations that can be called from other guardians, provide access to a guardians objects. Each guardian

resides at a single physical node, although a node may support many guardians. Guardians survive crashes of their node of residence and other hardware failures with high probability, and are therefore *resilient.* When a guardian's node crashes, all processes within the guardian are lost, but a subset of the guardian's objects, referred to as the guardian's *stable state,* survives. On recovery from a crash the guardian recovers from its saved stable state and runs a recovery process to reclaim the remainder of its objects. Resilience is accomplished in Argus by making a stable storage copy of the guardian's state periodically.

3 Language Features for Availability

3.1 Ad hoc techniques

Ad hoc techniques to implement availability are those in which the control of replication is programmed explicity in the object. These attempts at programming availability are normally inelegant and have no support from the system.

Master/Slave Appendix E of this report presents a detailed example of using one *ad hoc* technique—master/slave. In the master/slave paradigm (sometimes called the "hot spare" scheme), a designated replica performs all invocations and relays each invocation to the slave object. This keeps the slave's state completely up-to-date. An external object may hold a capability to the slave object as well as to the master, and may use this capability to perform a direct invocation on the slave; in this case, the slave merely passes the invocation immediately to the master; the invocation then proceeds as described above. All locks are obtained by the master, as allowing the slave to obtain locks concurrently could lead to deadlock. If the master fails, the slave will detect its unavailability when the slave attempts to relay an incoming direct invocation. In this case, the slave "promotes" itself to master, and creates a new slave object.

Although the *ad hoc* replication technique demonstrated by this example is inelegant, its use of the idea of having an object processing an invocation obtain the necessary locks at its replicas as well ultimately led to the development of the *Distributed Locking* scheme presented in Section 3.3. The advantage of Distributed Locking is that it requires no programming modifications to the single-site implementation of a resilient object in order to derive a replicated implementation.

3.2 Consensus Locking

Herlihy's work on General Quorum Consensus [Herl84] concerns the extension of quorum intersection methods to take advantage of the semantic properties of abstract data types. Previously, work on quorum methods—mostly in the database area—has been limited to a simple read/write model of operations. Herlihy's extensions allow the selection of optimal quorums for each operation of an abstract data type based on the semantics of that operation

and its interaction with the other operations of the data type.

Herlihy's method is based on the analysis of the algebraic structure of abstract data types. This entails the construction of a "quorum intersection graph," each node of which represents an operation of the data type, and each edge of which is directed from the node representing an operation *O1* to the node representing operation *O2*, where each quorum of *O2* is required to intersect each quorum of *O1*. From the quorum intersection graph, optimal quorums for each operation may be calculated, given the number of replicas of the data, and the desired availability of each operation in relation to the other operations of the data type.

Herlihy shows that his method can enhance the concurrency of operations on replicated data over that obtained from a read/write model of operations. He also claims advantages for his methods in the support of on-the-fly reconfiguration of replicated data, and in enhancing the availability of the data in the presence of network partitions.

More recently, Herlihy has developed two new methods for integrating concurrency control and recovery for abstract data types, called *Consensus Locking* and *Consensus Scheduling*.[5] In these schemes, Herlihy requires that the quorum intersection relation and the lock conflict relation (the complement of the lock compatibility relation) for an object satisfy a common *serial dependency relation* on that object; he notes that, in practice, the lock conflict and quorum dependency relations will be the same. [Herl85]

The model of objects and actions which Herlihy adopts may be summarized as follows.

- A replicated object is modelled as sets of *repositories*, which store the object state, and *front ends*, which perform operations for clients. An operation is performed by a front end by reading the object state from a collection of repositories, performing the computation, writing any update to the state to (another) collection of repositories, and sending a response to the invoker.

- In order to perform an operation on a replicated object, the front end must read from an *initial quorum* of repositories and must write to a *final quorum*, the sizes of which depend on the operation. Thus, a *quorum* for an operation is a set of repositories containing both an initial and a final quorum.

[5]The differences between these models need not concern us at present; therefore, we will refer to their common basis using the term "Consensus Locking."

- The object state is modelled as a *log*, which represents a *behavioral history* of the object, that is, a sequence of operation executions as well as *Commit* and *Abort* events. The log is formed by a sequence of entries consisting of a timestamp, an *event* (a paired operation invocation and response), and an action identifier. The log is partially replicated among the repositories, that is, some entries may be missing at some repositories.

- When a repository agrees to participate in an invocation, it grants an *initial lock*; when it agrees to accept a new log entry for an event, it grants a *final lock*. Both initial and final locks are two-phase.

- To execute an operation, the front end must merge the logs obtained from initial quorum of repositories. A *view* is constructed in which events belonging to committed actions are ordered by the actions' *Commit* entry timestamps, the invoker's operation is placed last, and entries belonging to uncommitted or aborted actions are discarded. The operation is then executed based on this view by a *single-site serial implementation* of the abstract data type.

Herlihy shows that Consensus Locking minimizes constraints on availability. On the other hand, the Consensus Scheduling scheme places additional constraints on availability, but allows more concurrency than Consensus Locking. While Consensus Locking is based on predefined lock conflicts, the Consensus Scheduling scheme allows scheduling decisions to take into account the state of the object.

A third scheme, called *Layered Consensus Locking*, extends the Consensus Locking method by associating a *level* with each activity in the system. [Herl85a] Activities at a higher level are serialized after activities at a lower level. If an activity executing at a given level is unable to make progress after a failure with its current quorum assignment, it may restart at a higher level and switch to another quorum assignment. Each initial quorum for an invocation at level n is required to intersect with each final quorum for an event at levels $\neq n$.

Herlihy and Wing recently have been developing a set of linguistic features, called *Avalon*, for support of transaction processing. [Herl87] Avalon is intended to be implemented as extensions to pre-existing languages such as Ada and C++. One aspect of this support closely resembles the features for support of action event handling provided by

Aeolus, as described in Section 1.3. Avalon also provides support for testing serialization orders dynamically.

3.3 Distributed Locking

3.3.1 Overview of Distributed Locking

In this section, we outline a model of concurrency control and replication management for the Clouds system, called Distributed Locking (**DL**). The linguistic and runtime mechanisms required to support DL are described in the following sections.

In the DL methodology, derivation of a replicated object from its single-site implementation consists essentially of two steps:

- The user writes a single-site definition and implementation of the object. This implementation includes specification of all lock types used by the object to ensure view atomicity in the presence of concurrently-executing actions.
- The user writes an *availability specification* (`availspec`) for the object. This specifies the number of replicas of each instance of the object to be generated, the replication control policies to be used, and (optionally) the relative availabilities of the modes of each lock type specified by the object. If no `availspec` is provided, the object is assumed to be nonreplicated.

The `availspec` construct is discussed in detail in Section 3.3.2. Note that availabilities are expressed in terms of the modes of locks rather than in terms of operations. Together with the *domain* notion, with which lock granularities are expressed in Aeolus/Clouds, this gives the user more latitude in the expression of relative availabilities than is provided in related work.

The automation of replication provided by the DL methodology is based on a concept similar to that of *action events* and *object events* as discussed in Section 1.3. The programmer may specify the interaction of an object with the action management system at critical points in the processing of an action via writing handlers for the action events; handlers for object events allow the object to participate in its creation and destruction. In a similar spirit, we have identified two critical points in the handling of an operation invocation on a

replicated object: the *lock event*, during which the invocation attempts to synchronize some subset of the replicas of the object; and the *copy event*, during which the state resulting from the invocation is transmitted to the subset of replicas synchronized during the corresponding lock event. These events correspond to the concurrency control and consistency maintenance aspects of replication control, respectively. Note that the names we have chosen for these events reflect the lock-based synchronization and stable storage-based recovery mechanisms of Clouds. For reasons examined in Section 5, we require that an invocation on a replicated object be made in the context of an action.

Policies for control of concurrency among replicas, and for control of the copying of state among replicas, are expressed in a `lock` object event handler and a `copy` action event handler, respectively, in the `availspec` for an object. Preprogrammed default handlers for these events, implementing commonly-used schemes such as quorum consensus, may be requested by the user if appropriate. If the user wishes to provide application-specific handlers for these events, the same system-provided primitives used in the construction of the default handlers are available for use in programming user-specified handlers. These primitives are described in Section 5, and example event handlers using the primitives are also presented there.

3.3.2 Availability Specifications

As discussed in Section 3.2, the Consensus Locking model of Herlihy allows the specification of the availability properties of an abstract data type in terms of the initial and final quorums required for an operation. It has already been mentioned that in the Distributed Locking model it makes sense to speak of the availability properties of lock modes (rather than of operations, as in other schemes). Some means is needed of allowing the programmer to specify these availability properties for an object without requiring modification of the single-copy version of the object definition or implementation.

In Distributed Locking as implemented in the Aeolus/Clouds system, the availability properties of a replicated object are specified in a separate compiland for that object type, called the *availability specification part* (or `availspec`, for short). The properties specified in an `availspec` include the number of replicas, the replication management algorithm desired (*e.g.*, quorum assignment, available-copies, etc.), the name of each lock type declared by the implementation of that object along with the names of that lock's modes, and (optionally)

the availability relationships among the modes of each lock type used by the implementation of that object. All internal and/or non-Clouds objects used by a replicated object must also have a replication specification; this requirement is applied recursively to these objects. The availability information of a non-Clouds object is inherited by the object which imports it; thus, the effect is as if locks declared by non-Clouds objects were instead declared by the importing Clouds object.

If a voting method is chosen, the quorum assignments for each lock may be derived from the replication specification using integer programming methods. The availability relationships among locking modes, expressed as relative availabilities, may be transformed into constraints on the space of feasible solutions; the objective function may be chosen to maximize the minimum availability over the locking modes subject to these constraints. The construction of this linear program is discussed in more detail later in this section. This information is transformed by the Aeolus compiler into a table of replication management information which is stored in the `TypeTemplate` of the given Clouds object. This information is placed in the header information of each object instance and is used by the Distributed Locking primitives to guide the selection of sets of replicas for Distributed Locking (see Section 5).

The Aeolus *availability specification* bears some resemblance to the *fault-tolerance specification* of the HOPS system (cf. Section 3.4). However, in HOPS the programmer must select among several predefined policies for replication control; there is no provision for user programming of these policies. The ability of the programmer to specify `lock` and `copy` event handlers as well as the provision of primitives in support of programming these handlers allows the use of a wider range of replication control policies with the Aeolus `availspec` construct.

Example of an Availability Specification A sample `availspec` making use of the `quorum` event handlers is given in Figure 5. This `availspec` applies to a resilient symbol table object, the definition for which is presented in Appendix B; the implementation of this object is presented and discussed elsewhere. [Wilk87] The degree of replication (*i.e.*, the number of replicas for a given instance of `symtab`) is given as a formal parameter to the `availspec`; the actual parameter is supplied (in addition to any object parameters specified by the definition part of the object) during creation of object instances.

```
availspec of object symtab ( d : unsigned ) is

   ! Availability specification of the symbol table object using
   ! the quorum consensus scheme.  The DistLock pseudo object
   ! definitions are imported automatically by all availspecs,
   ! but we must import the quorum definitions to use its
   ! predefined handlers.

   import quorum

   ! First, we specify the degree of replication (the number of
   ! replicas).  Here, the degree is taken from an additional
   ! parameter, d, which is specified during creation of an
   ! instance of this object.

   degree is d

   ! The resilient symtab object defines two locks, each with two
   ! modes.  We define the relative availabilities for the modes
   ! of each lock as follows.  The relative availabilities are
   ! used in the constraints of an integer program which is used
   ! in turn to generate the quorum assignments for each lock
   ! mode.

   lock symtable_lock with exact = nonexact

   lock name_lock with read > write

   ! The definitions of the lock and copy events.  Here, we just
   ! use the predefined handlers for quorum consensus.

   availspec events
      quorum_lock overrides lock_event,
      quorum_copy overrides copy_event

end availspec. ! symtab
```


Figure 6: Availability Specification for the Resilient Symbol Table

The **availspec** also specifies the relative availabilities of the modes of each lock declared by **symtab**. Here, the two modes of **symtable_lock** are declared to have the same availability level; however, the **read** mode of **name_lock** is declared to be more available than the **write** mode. The relative availability declarations are used to determine the size of quorums for each mode.

Finally, the alternate handlers for the **lock** and **copy** events are specified. Here, the **quorum_lock** and **quorum_copy** operations made accessible by importing the **quorum** pseudo-object are used.

Computing Quorum Assignments When a voting method is used for replication control, the system requires information about the minimum number of replicas required to constitute a quorum for each lock mode. As shown in the example **availspec** in the previous section, the programmer may specify the relative availabilities of the modes of each lock. This information is used to generate constraints for an integer program which computes the actual quorum requirements; the requirements for the modes of each lock of the object are then stored in the object state in an array associated with that lock. A primitive is provided for use in a **lock** event handler which returns the minimum quorum size associated with the lock and mode active at the invocation of the handler (that is, the request for which caused the **lock** event). The Distributed Locking primitives are described in Section 5.

The integer program used to generate the quorum information for each lock is built as follows. If the ith variable of the integer program represents the minimum number of replicas required to constitute a quorum for mode i of the lock, then the objective function is chosen to minimize the maximum value over all of the variables. As the availability of a mode is inversely proportional to the size of the quorum required for that mode, the objective function has the effect of maximizing the minimum availability over the modes. The relative availabilities of the locking modes as specified by the programmer in the **availspec** are used as constraints on the integer program; if no relative availabilities were specified, the availabilities of the modes are taken to be equal. There are additional constraints generated by the requirement of voting methods that the quorums of each pair of modes intersect (that is, that the sum of each pair of variables be greater than or equal to the degree of replication plus one), as well as that the value of each variable be nonnegative and be less than or equal to the degree of replication.

3.3.3 Comparison of Consensus Locking and Distributed Locking

In Section 1.3, the model of objects and actions provided by the Clouds system was described. The model used by Herlihy in his Consensus Locking scheme was described in Section 3.2. The salient differences between the Clouds action/object model as used by Distributed Locking and the model used by Consensus Locking may be summarized as follows.

- There is no logical separation between the object code (Herlihy's "front end") and the object data ("repository"), and thus no separation between reading the data, computation, and writing the data. An object operation performs computation interleaved with arbitrary examinations and modifications of the object state. A replicated Clouds object consists of a set of single-copy Clouds objects.
- Likewise, there is no separation of a quorum into initial and final quorums for an operation. Indeed, as will be explained below, quorums are not expliclitly associated with operations.
- Rather than using logs, the object state is kept in virtual memory, and its resilience is maintained by use of *shadowing* to perform atomic update. Different replicas of an object may have different *versions* of the object state at a given time. Each version is assigned a version number with which the most current version among the replicas may be identified. A good candidate for implementing the version number would be the timestamp of the Commit action event which creates the version (a usage similar to Herlihy's method of ordering log entries).
- There is no separation of "initial" and "final" locks. An operation may obtain a lock at an arbitrary point in its computation. Locks are normally two-phase, although the programmer may explicitly release locks before commit or abort if the desired consistency requirements permit.
- There is no necessity of merging logs into a *view*. An operation is performed by invoking one of the set of replicas of the desired object, each of which implements a single-site version of the object type. When a lock request is encountered in the invoked object instance, the Clouds kernel (using a modified naming scheme described in Section 5.1) attempts to set that lock on some subset of the replicas according to

the policy specified in the `lock` event handler. If the action performing the invocation has not previously touched the replicas, the most current version of the object state is identified, and this version is either copied to the replica executing the operation, or the execution may be transferred to a replica containing the most current version; this version becomes the action's view, which it may proceed to modify. If the action has already touched the replicas, then it already possesses its own view. The update of the replicas on action commit with the action's view is discussed below.

Another significant difference in Herlihy's work and that presented here is the model of locks. Locking is performed in Herlihy's model on an operation-by-operation basis; conflicts are defined among operations. Thus, in terms of Aeolus/Clouds Distributed Locking locks, one of Herlihy's Consensus Locking locks is defined with one locking mode per operation. There is no concept of the domain of a Consensus lock, as there is in Distributed Locking. Effectively, the domain of a Consensus lock is an entire object, *i.e.*, only one request for such a lock for a given operation is granted at a time, conflicts permitting. Thus, a Consensus lock for an object may be modelled by a Distributed Locking lock with one mode per operation and no domain. However, by allowing specification of arbitrary modes and domains, Distributed locks allow more generality than Consensus locks. The programmer may decide to share some lock modes among operations based on semantic similarities between those operations (for instance, examine vs. modify operations), thus effectively defining *classes* of operations with similar concurrency and availability characteristics. It is also possible that the programmer may decide to have an operation obtain a lock in different modes depending on its parameters or other factors; this may occur, for instance, through consolidation of logically separate operations with a similar interface into a single operation (to avoid duplication of portions of their functionality). Thus, while it is reasonable in Consensus Locking to speak of the differing availabilities of *operations* rather than of objects, it is also sensible to speak in Distributed Locking of the availability of *lock modes*. In addition, the ability to specify a domain for an Aeolus/Clouds lock may permit increased concurrency over locking on the object operation level.

3.3.4 Built-in Replication Schemes in Distributed Locking

Several popular replication management schemes are available when using the Distributed Locking mechanism. Other schemes can be supplied by the programmer.

Primary copy The primary copy methods have multiple copies of the data around the network with one designated copy recognized as the "primary copy". These methods insist that access to a copy during any network partition is allowed, only if the partition possesses the designated primary copy of the data. If the primary copy is not in the partition, no access is allowed until the failure(s) have been corrected allowing access to the primary copy.

Token passing scheme An extension of the primary copy method, the token passing scheme passes a token among the various sites on the network holding a copy of data. The copy at the site currently holding the token is considered the primary copy.

Voting schemes Voting schemes are another extension of the primary copy method. Each copy of the data object is assigned a number of votes. The number of votes for each copy may vary from copy to copy. The partition which possesses the majority of the votes for the object may access it.

Available copies Available-copy methods follow a "read-one, write-all available" discipline. A **read** operation may access any *initialized* copy (that is, one which has already processed a **write** operation). A **write** operation must access all copies; those which are unavailable for writing are called *missing writes*. A validation protocol, which runs after all **reads** and **writes** of a transaction have either been processed or timed out, guarantees one-copy serializability. This protocol ensures that all copies for which missing writes were recorded are still unavailable, and that all copies accessed are still available. Several researchers have recently proposed enhancements to the original available-copies algorithm. [Skee85] [El-A85] [Long87]

General quorum consensus See section 3.2.

3.3.5 Distributed Locking Example

We will now consider an example of distributed locking using the resilient symtab object with quorum consensus handlers.

Assume the capability S refers to (some convenient replica of) the replicated resilient symbol table object. (The choice of replica will be made by the system depending on the naming scheme.) The replica chosen is called the primary cohort, or p-cohort.

- object O invokes operation S@insert(name, value) at the p-cohort in the context of an action A.

- the S@insert(name, value) operation obtains the name_lock in write mode on the value name. This causes a lock event, so the lock event handler for the replicated symtab type is invoked. In this case, the event handler implements quorum consensus, so it attempts to obtain the same mode and value of name_lock at some quorum of replicas of S (including the p-cohort). The quorum size is determined by the relative availabilities of the modes of name_lock. If a quorum can't be obtained, action A is aborted. The members of the set of replicas other than the p-cohort that belong to the quorum are called the s-cohorts.

  Note that the insert() operation later obtains another lock, the symtable_lock (in mode nonexact). The quorum size needed for this lock is determined by the relative availabilities specified for its modes, which may be different than the quorum size needed when we obtained the name_lock. If the size is larger, we must obtain locks at additional replicas to bring the quorum obtained earlier up to the new size. If it's smaller, we stick with the larger quorum required by name_lock.

- When action A commits, a copy event is caused, so the copy event handler is invoked. The event handler for quorum consensus causes the state of the p-cohort to be copied to the s-cohorts. The copied state is then committed at those replicas.

  If action A aborts, the previous state of the p-cohort is restored (as in a single-site Clouds object), and the locks obtained at the s-cohorts are released. As no state was copied to the s-cohorts prior to end of action, nothing needs to be restored there.

3.4 Other Work

3.4.1 The HOPS project

The *Honeywell Object Programming System* (**HOPS**) [Hone86] under development at Honeywell, Inc., has research goals similar to those of our methodology research. The stated

goals of the HOPS project are:

- to alleviate what is seen as a lack of experience in the field of distributed systems in implementing mechanisms which perform failure detection, failure recovery, and resource reconfiguration;
- to provide programming support for developing fault-tolerant distributed applications; and
- to assess the actual benefits and costs of such mechanisms in terms of performance, reliability, and availability.

HOPS consists of an implementation language derived from Modula-2 together with a distributed runtime support system. The language requires that HOPS objects (or *HOPjects*) be specified in three parts: an interface specification, a body (or implementation specification), and a *fault tolerance specification.* In the latter, the programmer may specify attributes and policies relating to recovery, concurrency control, and replication which are to be used for that object, thus giving the programmer a choice among several mechanisms provided by HOPS in each of these areas. The distributed runtime system (together with the underlying host operating system) provides facilities for naming and addressing objects, communication, failure detection and recovery, local and distributed transaction management, concurrency control, recovery, and replication. HOPS is currently being implemented on a network of Sun-3 workstations under the Sun version of Unix 4.2.

4 Support for resilience

The features provided by Aeolus to access the support for resilient computation provided by the Clouds system were described in Chapter II. These features rely heavily on the action management subsystem of Clouds for their functionality. In this section, the interface to the action manager is described, as is other support for synchronization and recoverability required of the Aeolus compiler.

4.1 The Action Manager Interface

The Aeolus interface to the Clouds action management subsystem is presented in its entirety in Appendix G. This interface is implemented as an Aeolus pseudo-object called `ActionManager`. Operations are provided to:

- return the action ID, status, and level (top-level or nested) of the current action;
- query and set the timeout of a specified action;
- obtain the current global Lamport clock time;
- commit the current action;
- abort the current action or a specified child action; and
- await the completion (commit or abort) of a specified child action.

Operations used by the language runtime system, but not available to the programmer in the Aeolus/action management interface, are those for creating actions. The interface to the `AM_Create_Action` and `AM_Create_Proc_Action` operations are described in Chapter III of Kenley's thesis. [Kenl86] `AM_Create_Action` is used to invoke an operation of a remote Clouds object as an action, while `AM_Create_Proc_Operation` is used to invoke a local procedure of an object as an action.

4.2 The Clouds Object Header

Each Clouds object is provided with a header containing information used by both the object and action management subsystems. This header is generated automatically by the

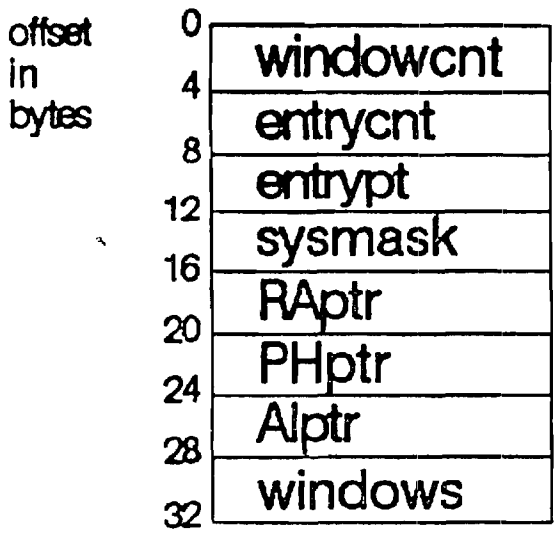

Figure 7: Clouds Object Header

Aeolus compiler from the object type definition and implementation. The layout of the object header is illustrated in Figure 7.

The fields of the object header are as follows:

windowcnt The number of "windows" used by the object. A window is a region of memory with certain associated protection characteristics (*e.g.*, read-only or execute-only). Thus, code, data, recoverable areas, and the permanent heap are typically in separate windows.

entrycnt The number of public operations provided by the object.

entrypt The address of the common object entry point for the public operations (called `ObjEntry` in the Aeolus runtime system). Any required transformations of operation parameters (*e.g.*, from kernel to Aeolus format) are performed by this entry point.

sysmask A bit mask indicating which "system operations" (object event handlers) are provided by the object.

RAptr The address of the beginning of the recoverable area of the object, if any.

PHptr The address of the beginning of the permanent heap area of the object, if any.

AIptr The address of the "availability information" tables containing the minimum quorum size information for each lock mode, if the object has an availability specification.

windows The address of the *window descriptors* for each window used by the object. The format of window descriptors is described in Spafford's dissertation. [Spaf86]

More details on runtime support for objects are provided in Appendix H.

5 Support for availability

5.1 Support for Distributed Locking

As defined in Section 3.3, the term Distributed Locking refers to a *methodology* for deriving a replicated implementation from its single-copy version, as well as to a *mechanism* to support this methodology. A powerful feature of Distributed Locking is that it does not assume any particular *policy* for replication control. Although the user may easily specify use of one of several default policies in the areas of replica concurrency control and state copying, it also allows the user to explicitly program policies for these purposes. The mechanisms provided by Distributed Locking for support of both default and user-programmed policies are described below.

5.1.1 Naming Replicated Objects

The mechanism required for support of Distributed Locking requires modifications to the Clouds object naming scheme to support replication.

We have considered two different capability-based naming schemes which may be used in support of *state cloning*, as described in Section 1. The first scheme requires minimal changes to the Clouds kernel, but relies on facets of the Clouds object lookup mechanism which may not be applicable to other systems. In Clouds, the search for an object begins locally (that is, on the node which invoked the search), and—if the object is not found locally—proceeds to a broadcast search. If the internal objects belonging to a replica are constrained to reside on the same node as their parent object, then the local search will locate the local instance of the internal object. (This constraint is not considered to be onerous, since the internal objects of each replica need to be highly available to that replica in any case, and thus should logically reside on the same node as the parent replica.) Thus, each replica of an object (each of which resides on a separate node) may maintain its set of internal objects using the same capabilities as each other replica. (This situation may be created by initializing one replica, and then cloning its state to the other replicas.) Although there will thus be multiple instances (on separate nodes) of internal objects referenced by the same capability, there should be no problems caused by this, since—by the definition of internal object—only the parent object or its internal objects may possess the capability to an internal object, and the object search will always locate the correct (local) instance.

Thus, state cloning may be used to copy the state of a replica to the other replicas without causing the problems with respect to internal objects described in Section 1 (concerning references to internal objects contained in the replica's state), since under this scheme all replicas may use the same capabilities for referencing internal objects. This scheme is an extension of a facility already supported by the Clouds kernel for cloning read-only objects such as code. This scheme is called *vertical replication*, since it maintains the grouping of internal objects with their parent object.

The other naming scheme makes fewer assumptions about the lookup mechanism than vertical replication, but requires more kernel modifications. In the second scheme, each instance of the replicas' internal objects is again named by the same capability, at least as far as the user is concerned; however, the kernel maintains several additional bits associated with each capability identifying a unique instance. (These additional bits may be derived, for example, from the birth node of the instance.) When a (parent) replica invokes an operation on an internal object, the kernel selects one of the replicas of the internal object according to some scheme (*e.g.*, iteration through the list of nodes containing such objects until an available copy is located). Thus, a set of replicas of internal objects is maintained in a "pool" for access by all parent replicas. Again, each parent appears to use the same (user) capability to reference a given internal object, so the problems of state cloning disappear. Since this scheme maintains a logical grouping of the copies of an internal object, rather than grouping internal objects with their parent object, this scheme is called *horizontal replication.* One such naming scheme is described in a paper by Ahamad *et al.* [Aham87]

The attractions of the vertical replication scheme are that it is conceptually simple, that it requires no modifications to the kernel capability-handling mechanisms, and that, by requiring coresidence, it enforces a property which enhances availability. To see this, recall that independent failure modes are desirable among different replicas of a replicated object, since the probability that the replicated object will be available is the probability that *any one* of the set of replicas will be available. On the other hand, dependent failure modes are desirable among a given replica and its internal objects, since the probability that the given replica will be available is the probability that *all* of the set of internal objects will be available. Requiring coresidence of objects related by logical nesting introduces dependence of their failure modes.

Unfortunately, the vertical replication scheme is not viable in general, since the coresidence requirement may sometimes be unrealistic. It may sometimes be the case that it is

impossible to satisfy coresidence, due to the size of nested objects (making it impossible to accommodate them on the same node), or due to insufficient space because of previously-existing objects on that node. Thus, vertical replication must be abandoned as lacking sufficient generality in its applicability. Fortunately, the horizontal replication scheme does not share this drawback.

The horizontal replication scheme has been further developed in a recent paper by other researchers on the Clouds project. [Aham87a] However, the invocation scheme may be altered to take advantage of coresidence when possible. The search scheme used for invocation of replicated objects in the paper cited above involves a random choice among the set of replicas. This differs markedly from the current Clouds search scheme for non-replicated objects, which is essentially as follows:

```
if <object found locally> then
   <perform invocation on local object>
else
   <perform global search>
end if
```

This search scheme may be modified to take advantage of coresidence as follows:

```
if <object found locally> then
   <perform invocation on local object>
else
   if <object is replicated> then
      <select randomly among the set of replicas>
   else
      <perform global search>
   end if
end if
```

Note that, if only one replica is stored per node, the local search involves only the so-called "user capability;" that is, it does not involve the extra bits used by the "kernel capability" to distinguish among replicas. If one allows more than one replica per node, some use of the kernel capability must be made to select an appropriate instance; this may require specific knowledge of which replicas are stored at which nodes.

5.1.2 Invocation of Lock and Copy Events

Support of the Distributed Locking mechanism requires modification of the Aeolus/Clouds object and action management facilities in two areas.

- When an operation attempts to obtain a lock on an instance of a replicated object, locks are obtained at some appropriate subset of its replicas, by invoking the *lock* event handler on that object. (Using terminology introduced by Ahamad and Dasgupta, [Aham87a] the replica at which the original invocation took place is called the *primary cohort* [*p-cohort*]; the other members of the locked subset of replicas are called *secondary cohorts* [*s-cohorts*].)
- During the handling of the precommit event of the controlling action, the state of each p-cohort touched by that action is copied to its s-cohorts, by invoking the *copy* event handler on each p-cohort.

In Section 1, two methods of copying object state applicable to the Clouds model were identified:

- *idemexecution*, or execution of an invocation at each member of the set of replicas; and
- *cloning*, or execution of an invocation at a single replica, and then explicitly copying its state to the other replicas.

Because of the drawbacks of idemexecution (including the possibility of repeated invocations on objects external to the replicated object, as well as the difficulty of handling invocations with non-deterministic results in this scheme), the most viable mechanism seems to be cloning. However, the Distributed Locking mechanism does not preclude the use of idemexecution in the *copy* event, and provides primitives for its support.

Since a replicated object may have an arbitrary structure of logically nested objects, it is a non-trivial problem to determine exactly what state of which objects must be copied to implement a cloning operation. That is, it does not suffice to merely copy the state of the *p-cohort* to its *s-cohorts*; the states of all objects nested with respect to the *p-cohort* which were involved in the given operation must also be copied to their respective replicas

(the nested objects of the *s-cohorts*). Fortunately, the Clouds action mechanism provides a means of determining which objects must be cloned: the action manager maintains a list of objects *touched* by an action. (This is the reason behind requiring that invocations on replicated objects take place in the context of an action.) Indeed, one need only perform cloning upon commit of an action, since the results of an action become visible to other actions only after commit. At that time, the so-called "shadow set" of each touched object is available. (In very simplified terms, this is the set of pages in the object's recoverable area which have been modified by the action.) If the constraint is made that all replicated objects be recoverable, then to implement cloning, one need only copy the shadow set of each touched object to the other replicas in that object's set, and perform the commit actions of storage management at each replica. The shadows are committed at each of the s-cohorts as if the shadows had been produced by execution at that s-cohort.

5.1.3 Primitives for Lock and Copy Event Handlers

If the user wishes to provide application-specific handlers for these events, the same system-provided primitives used in the construction of the default handlers are available for use in programming user-specified handlers. These primitives, and their purposes, include those for such purposes as:

- acquisition at a specific replica of the currently-requested lock (with the same mode and value, if any), for implementing lock propagation;
- invocation at a specific replica of the same operation (with the same parameters) requested at the current replica, for implementing idemexecution;
- broadcast of state shadow sets to all replicas holding a specified lock (with a specified mode and value), for implementing cloning via shadows; and
- invocation at a specific replica of an arbitrary operation, for implementing cloning via logs or state reconciliation strategies.

The intention is to provide facilities at a level sufficiently low to accommodate all schemes of interest. Some other useful predefined objects, such as those implementing list abstractions, are available for such purposes as maintaining and traversing the list of replicas at which locks have been obtained (and to which the object state must later be copied).

The primitives described above are encapsulated in an Aeolus pseudo-object called *DistLock*. The definition of *DistLock* is presented in its entirety in Appendix F.

```
implementation of pseudo object quorum is

   ! Here, we define handlers for the lock and copy events which
   ! implement quorum consensus.  This pseudo object is imported
   ! by any availspec wishing to use its predefined handlers.

import DistLock

procedure quorum_lock () is
   ! A simple-minded lock event handler for quorum consensus.
   ! Locks are obtained on at least a minimum quorum assignment
   ! specified by the assignment matrix generated by the
   ! importing availspec.

   this_version ,
   max_version  : version_number
   num_locked   ,
   good_replica : replica_number

   begin
      ! Find out how many replicas have been locked already by
      ! the current action.
      num_locked := DistLock @ currently_locked()

      ! Initially, the latest version seen is set to this
      ! instance's version number.
      max_version  := DistLock @ my_version()
```

```
   ! Attempt to lock all available replicas.
   for r in replica_number[ 1 .. DistLock @ degree() ] loop
      if DistLock @ lock_replica( r, this_version ) then
         num_locked += 1
         if this_version > max_version then
            max_version  := this_version
            good_replica := r      ! remember the latest version
         end if
      end if
   end loop

   ! At least a quorum of replicas must have been locked.  If
   ! not, abort the invoking action.
   if num_locked < DistLock @ quorum_size() then
      Abort_Myself()
   end if

   ! If there is a later version of the state than that of
   ! this replica, copy it here.  (This updates the local
   ! version number.)
   if good_replica <> DistLock @ my_replica() then
      if not DistLock @ get_state( good_replica ) then
         Abort_Myself()  ! replica was unavailable
      end if
   end if

   ! Copy the local state to all replicas which have version
   ! number less than that of the local copy.
   for r in replica_number[ 1 .. DistLock @ degree() ] loop
      if not DistLock @ send_state( r ) then
         Abort_Myself()  ! replica was unavailable
      end if
   end loop
end procedure ! quorum_lock
```

```
procedure quorum_copy is
   ! The copy event handler for quorum consensus.  The shadow set
   ! is copied to the set of replicas locked in the lock event.

   begin
      if not DistLock @ broadcast_shadows() then
         Abort_Myself()  ! copy was unsuccessful
      end if
   end procedure ! quorum_copy

end implementation. ! quorum
```

Figure 8: Lock and Copy Event Handlers for Quorum Consensus

5.1.4 Examples of Event Handlers in Distributed Locking

A sample implementation of **lock** and **copy** event handlers using the General Quorum Consensus algorithm are given in Figure 6. The treatment of these event handlers has been kept on a fairly naive level to avoid obscuring neither the general lines of the algorithm used nor the use of the Distributed Locking primitives. The handlers are encapsulated in a pseudo-object called **quorum** which may be imported by an **availspec** in order to use its handlers.

As described in a previous section, the replica of an object at which an operation is invoked is called the *primary cohort* or *p-cohort*; a request for a lock at the p-cohort causes its lock event handler to be activated. The handler for the **lock** event, here called **quorum_lock**, attempts to lock each other available replica (called *secondary cohort* or *s-cohort*) by use of the **lock_replica** Distributed Locking primitive; if successful, this primitive returns the version number of the new s-cohort as an **out** parameter. The maximum version number over all s-cohorts is determined and compared with the version number of the p-cohort; if the latter is not the latest version, the state of the s-cohort having the latest version is copied to the p-cohort. In any case, at this point the latest state is copied to all s-cohorts having earlier states. If the number of s-cohorts is not at least as great as the quorum

assignment for the requested lock mode, the enclosing action is aborted.

When the action enclosing the operation invocation prepares to commit, the copy event handler (here called quorum_copy) is activated. This handler uses the broadcast_shadows primitive to copy the shadow set (of changed pages) of the p-cohort to the s-cohorts locked in all activations of the lock event handler by the current action. If the copy is successful, the shadow sets are committed at the s-cohorts as well as the p-cohort to yield the updated state.

There are obvious improvements which might be made to this simple version of quorum. For example, quorum_lock relies on the lock_replica primitive to "fall through" when an attempt is made to lock a replica which is already an s-cohort. A more sophisticated implementation could maintain a set of replica numbers representing the current set of s-cohorts in order to avoid the overhead of a remote invocation for each redundant lock_replica call.

The use of the broadcast_shadows primitive in quorum_copy requires that the states of all s-cohorts be identical to that of the p-cohort when the lock event handling is complete, so that the shadow set broadcast during the copy event can be committed into a common permanent state at each replica; this is achieved by copying the state of the replica with the latest version number to those replicas with earlier versions of the state. This implementation assumes that it is uncommon for the version number of a replica to be "out of synch" with its fellow replicas, which is a reasonable assumption if most, if not all, replicas are available to become s-cohorts during each lock event. If this assumption is invalid, it may be more efficient to avoid copying of the latest state to the s-cohorts during the lock event *and* copying shadow sets during the copy event by copying the entire state of the p-cohort to the s-cohorts during the copy event.

A Permanent Heap Example

This appendix contains the Aeolus source code for an example permanent heap object.

```
definition of recoverable object permheap is
   ! Gives the publically-visible definitions provided by the
   ! permheap object.

operations

   procedure allocate ( size : unsigned )
                        returns address modifies
      ! Return a pointer to a block of memory of the given
      ! size (in words) in permanent memory.

   procedure free ( block : address ) modifies
      ! Dispose the block of memory indicated by block.

end definition. ! permheap !

implementation of ! recoverable ! object permheap is
   ! Support for the permanent heap, using per-action variables
   ! for recovery management.

import list

! The definition part of the LIST object is shown here for
! clarity.
!
! definition of local object list ( elem_type: type ) is
!    -- This object implements a linked list abstraction.
!    -- The object is parameterized by the element type of
!    -- the list; if the element type is specified to be
!    -- permanent by a (recoverable) importing object, then
```

```
!    -- the linked list itself will be allocated in
!    -- permanent storage (only recoverable objects may
!    -- declare permanent variables).  The list is
!    -- initially empty.  Mutual exclusion is provided on
!    -- \f6modifies operations.

! operations
!    procedure add ( elem: elem_type ) modifies
!       -- Adds elem to the list.
!    procedure append ( l: list ) modifies
!       -- Appends all elements in list l to this
!       -- list.  Use of the object type list here
!       -- with no parameters implies that list l must
!       -- have the same element type as this list.
!    procedure remove ( elem: elem_type ) modifies
!       -- If elem is on the list, removes it.
!    procedure find ( elem: elem_type ) returns boolean examines
!       -- If elem is on the list, returns TRUE,
!       -- otherwise FALSE.
!    procedure nth ( n: unsigned,  notthere: out boolean )
!                  returns elem_type modifies
!       -- If the nth element exists, returns it and
!       -- sets notthere to FALSE, otherwise sets
!       -- notthere to TRUE.
! end definition. -- list

! The local declarations of the permheap object.
!
! Here, we give the names of alternate handlers for some of the
! action events.  Note that no alternate handler is given for the
! abort event (see section 6).

action events
            permheap_boa overrides boa,
```

```
          permheap_commit overrides commit,
    permheap_top_precommit overrides toplevel_precommit

!-----------------------------------------------------------------
! The perm_blockentry type is used for the maintenance in
! the permanent heap of the list of free storage blocks.  Each
! block is uniquely identified by its address.

type perm_blockentry is permanent new address

!-----------------------------------------------------------------
! The list of free storage blocks.  Since the base type of this
! list is permanent, the list itself is allocated in permanent
! storage.  This list may be modified only during the
! toplevel_precommit action event.  The size of each entry is
! stored in the first word of that entry.

freelist : permanent list( perm_blockentry ) := new list

!-----------------------------------------------------------------
! The blockentry type is used in the declaration of the
! per-action variables below.  Pointers to this type are
! allocated on the normal (not the permanent) heap, and may be
! modified outside of the toplevel_precommit event handler.

type blockentry is new address

!-----------------------------------------------------------------
! The per-action variables for permanent-heap recovery
! management.  We will maintain lists of memory blocks allocated
! and freed by each action.  These lists are initialized in the
! handler for the BOA action event.

per action
```

```
   allocated, freed : list( blockentry )
end per action

!-------------------------------------------------------------
! When an action allocates a block of permanent storage, it must
! obtain a lock on that block until it commits to prevent other
! actions from attempting to allocate that block.  Rather than
! associate a lock with the actual storage block, we lock the
! block's address (of type blockentry).  Recall that locks
! obtained by an action are propagated to its parent upon nested
! commit, and released upon abort or toplevel commit.

entry_lock : lock ( busy : [] ) domain is blockentry

procedure first_fit ( size : unsigned ) returns blockentry is
   ! A private operation of the permheap object.  Given a
   ! size in words, first_fit finds the first entry on the
   ! freelist for a block of storage of size at least as
   ! large as size and returns a pointer to that entry.
   ! (For the purposes of this example, we will assume that such
   ! a block exists.) Of course, another strategy could also be
   ! used here (such as best fit, or fragmentation and
   ! compaction).  We'll assume that repeated invocations of
   ! first_fit by the same action return different
   ! addresses.

   begin
      ! The details of this operation are omitted here.  Even if
      ! an appropriate block of storage is found on the
      ! freelist, first_fit must also test the
      ! entry_lock to check whether this block has not
      ! already been allocated by some as yet uncommitted action.
   end procedure ! first_fit !

! allocate and free are public operations of the
```

```
! permheap object.

procedure allocate (! size : unsigned !) ! returns address ! is
   ! Return the address of a block of memory of the given
   ! size in permanent storage.  Since the block is from
   ! the freelist, its former contents are expendable.  The
   ! Set_Lock operation used here is non-blocking, i.e., it
   ! returns immediately with value FALSE if the requested lock
   ! is not available.

   entry : blockentry

   begin
      loop   ! keep going until we find an available block
         entry := first_fit( size )
         if Set_Lock( entry_lock, busy, entry ) then
            ! add the entry to the ALLOCATED list for this
            ! action
            Self.allocated @ add( entry )
            return address( entry )
         end if
      end loop
   end procedure ! allocate !

procedure free (! block : address !) is
   ! Add a block of memory to the freed list for
   ! freeing during toplevel precommit.

   entry : blockentry
   notthere : boolean
   i : unsigned := 1

   begin
      ! First, scan the allocated list to see if
```

```
      ! block was allocated by the current action
      loop
         entry := Self.allocated @ nth( i, notthere )
         if notthere then
            exit .
         elsif entry = blockentry( block ) then
            ! Yes, so remove it from allocated list
            Self.allocated @ remove( entry )
            ReleaseLock( entry_lock, busy, entry )
            return .    ! we're done
         end if
         i += 1
      end loop

      ! If we get here, block wasn't allocated by the
      ! current action, so put it on the freed list
      Self.freed @ add( entry )
   end procedure ! free !

! The following are the alternate action event handlers for this
! object.

procedure permheap_boa () is
   ! The alternate handler for the BOA (beginning of
   ! action) event.  The initial states of the per-action
   ! variables for the current action are set to be the same as
   ! those of its immediate ancestor; the effect is that the
   ! child action may view modifications made by its parent
   ! action before the child action was created.

   status : action_status
   level : action_level

   begin
```

```
      ! see if we're in a nested action
      Void( ActionManager @ Tell_ID( status, level ) )
      if level = nested_action then ! copy the parent's variables
         Self.allocated := ObjCopy( Parent.allocated )
         Self.freed     := ObjCopy( Parent.freed     )
      else ! this is a top-level action; allocate empty lists
         Self.allocated := new list
         Self.freed     := new list
      end if
   end procedure ! permheap_boa !

procedure permheap_commit () is
   ! The alternate handler for the commit action event.
   ! We'll propagate the items on the allocated and
   ! freed lists of this action to the corresponding lists
   ! of its parent action.

   status : action_status
   level : action_level

   begin
      ! see if we're in a nested action
      Void( ActionManager @ Tell_ID( status, level ) )
      if level = nested_action then
         Parent.allocated := ObjCopy( Self.allocated )
         Parent.freed     := ObjCopy( Self.freed     )
      end if
   end procedure ! permheap_commit !

procedure permheap_top_precommit () is
   ! The alternate handler for the toplevel_precommit
   ! action event.  We'll traverse the freed list, adding
   ! each entry there to the actual freelist in permanent
   ! storage; then, we'll traverse the allocated list,
```

```
   ! removing each entry there from the freelist.

   entry : blockentry
   notthere : boolean
   i : unsigned := 1

   begin
      ! Add each entry on the freed list to the
      ! freelist in permanent storage
      loop
         entry := Self.freed @ nth( i, notthere )
         if notthere then
            exit .
         end if
         ! Convert the entry to the permanent type before adding
         ! to freelist.
         freelist @ add( perm_blockentry( entry ) )
      end loop

      ! Remove each entry on the allocated list from the
      ! freelist; the locks on these entries will be
      ! released automatically.
      loop
         entry := Self.allocated @ nth( i, notthere )
         if notthere then
            exit .
         end if
         freelist @ remove( perm_blockentry( entry ) )
      end loop
   end procedure ! permheap_top_precommit !

inithandler is   ! handler for the initialization object event
   begin
      ! Perform initialization (not shown) of freelist to
```

```
      ! indicate that all of the permanent heap is available.
   end inithandler

!
! The DELETE object event handler for this object is by default
! NULL.
!

end implementation. ! permheap !
```

B Resilient Symbol Table Definition

This appendix contains the Aeolus source code for the definition part of a resilient symbol table object.

```
implementation of object symtab
      !( name_type : type,  value_type : type )! is

! Single-copy symbol table object using the lock mechanisms of
! Aeolus/Clouds for synchronization and to ensure view atomicity.
! This implementation of the symbol table uses the recoverability
! features of Clouds to provide resiliency.  The use of
! per-action variables to maintain ''intention lists'' of entries
! inserted or deleted during an action also helps ensure view
! atomicity, since each action gets its own version of the
! per-action variables.  Since this object is recoverable, we
! will not explicitly release locks; rather, when a lock is
! obtained by a nested action, it will be propagated to the
! immediate ancestor when the nested action commits, and will be
! released when the top-level ancestor commits.  The symbol table
! structure and its entries are kept in permanent storage.  Since
! permanent storage may be altered only at toplevel precommit, we
! maintain two ''intention lists'' of non-permanent entries which
! contain those entries which are inserted or deleted by an
! action.  The entries in these lists will be transferred to the
! permanent symbol table during toplevel precommit.

import list, keyed_list

! Here, we give the names of alternate handlers for some of the
! action events.  Note that we need not override the abort event.

action events
            symtab_boa overrides boa,
```

```
          symtab_commit overrides commit,
    symtab_top_precommit overrides toplevel_precommit

! The per-action variables for the symbol table are where we
! maintain the ''intention lists'' of entries inserted and deleted
! by an action.  The inserted list entries are keyed on the
! name field, but also include the value field.  The deleted
! list entries need merely give the name field.  These variables
! are initialized in the handler for the BOA action event.

per action
   inserted : keyed_list( name_type, value_type )
   deleted  : list( name_type )
end per action

! Each bucket of the hash table is a list of names and values,
! keyed by the name field.  The list objects are kept in
! permanent storage, and thus modify operations on them may be
! invoked only during toplevel precommit.  (However, examine
! operations may be invoked at any time.)

type bucket_list is permanent new keyed_list( name_type, value_type )

! The symbol table structure itself is an array of bucket lists.
! The array is also kept in permanent storage, and may be altered
! only at toplevel precommit.  Since action management ensures
! that only one action may be in the toplevel precommit handler
! at a time, there is no need to explicitly enforce mutual
! exclusion on the symbol table buckets, as is done in the
! nonrecoverable version of the symtab object by means of
! critical regions.

MAXBUCKET : const integer := 101     ! or whatever
```

```
type hash_range is new unsigned[ 1 .. MAXBUCKET ]

symtable : permanent array[ hash_range ] of bucket_list

! The symtable_lock allows the entire symbol table to be locked.
! This lock is set (in exact mode) in the exact_list operation for
! purposes of getting an exact listing of the state of the symbol
! table.  Operations which change the state of the symbol table
! must wait for completion of any outstanding exact_list
! operations and vice versa.

symtable_lock : lock ( exact    : [ exact    ] ,
                       nonexact : [ nonexact ] )

! The NAME lock allows the user to lock the name which is to be
! used in one of the symbol table operations.  The purpose of
! this lock is to assure the view atomicity of these operations,
! that is, to provide synchronization such that concurrent users
! of the symbol table do not view the results of other actions
! before those actions are committed.

name_lock : lock ( write : []     ,
                   read  : [read] ) domain is name_type

procedure hash ( name : name_type ) returns hash_range is
   ! This hash function is a local (nonpublic) procedure of the
   ! symtab object.

   begin
      NULL   ! the usual type of stuff
   end procedure ! hash

procedure sym_find ( name  : name_type      ,
                     value : out value_type ) returns boolean is
```

```
   ! The sym_find routine is a local (nonpublic) procedure of the
   ! symtab object.  It assumes that the caller has obtained the
   ! necessary locks.

   begin
      return    Self.inserted @ find( name, value )
             or (     not Self.deleted @ find( name )
                  and symtable[ hash( name ) ] @ find( name, value ) )
   end procedure ! sym_find

procedure insert (! name  : name_type    ,
                  ! value : value_type   ,
                  ! error : out boolean !) is
   ! The insert operation adds an entry to the inserted list for
   ! this action, if the entry is not found; otherwise, error is
   ! set to TRUE.  The entry will placed into the permanent
   ! symbol table at toplevel precommit.

   dummy : value_type

   begin
      Await_Lock( name_lock, write, name )
      error := sym_find( name, dummy )
      if not error then
         Await_Lock( symtable_lock, nonexact )
         Self.inserted @ add( name, value )
      end if
   end procedure ! insert

procedure delete (! name  : name_type    ,
                  ! error : out boolean !) is
   ! If the delete operation finds an entry with value field =
   ! name, it adds the entry to the deleted list; otherwise,
   ! error is set to TRUE.  The entry will be deleted from the
```

```
    ! permanent symbol table at toplevel precommit.

    dummy : value_type

    begin
       error := FALSE
       Await_Lock( name_lock, write, name )
       if Self.inserted @ find( name, dummy ) then
          ! If this action has inserted the name, it must already
          ! have a nonexact lock on the symbol table.  In this case,
          ! Await_Lock() would just return immediately, since we
          ! already have the lock.  Therefore, we won't bother
          ! invoking Await_Lock().
          Self.inserted @ remove( name )
       else if symtab[ hash( name ) ] @ find( name, dummy ) then
          Await_Lock( symtable_lock, nonexact )
          Self.deleted @ add( name )
       else
          ! name not in the permanent symbol table or inserted by
          ! this action
          error := TRUE
       end if
    end procedure ! delete

procedure lookup (! name  : name_type    ,
                  ! error : out boolean !) ! returns value_type ! is
    ! The lookup operation sets a read lock on the name entry, and
    ! then tries to locate that entry with name field = name and
    ! returns its value if it succeeds.

    value : value_type

    begin
       Await_Lock( name_lock, read, name )
```

```
      Await_Lock( symtable_lock, nonexact )
      error := not sym_find( name, value )
      return value
   end procedure ! lookup

procedure quick_list () is
   ! The quick_list operation provides a quick (dirty) listing of
   ! names currently in the symbol table.

   begin
      ! First, display the stuff in the permanent symbol table
      for i in hash_range loop
         symtable[ i ] @ display()
      end loop

      ! Now, display entries added by this action or its
      ! children, if any
      Self.inserted @ display()
   end procedure ! quick_list

procedure exact_list () is
   ! The exact_list operation provides a listing of the exact
   ! state of the symbol table at a given point in time.  To do
   ! this, it locks the whole symbol table, thereby excluding any
   ! changes during preparation of the listing.  Thus, although
   ! exact_list, lookup, and quick_list operations may execute
   ! concurrently, and insert and delete operations which access
   ! different hash buckets may also execute concurrently, insert
   ! and delete operations must block on exact_list operations
   ! and vice versa.

   begin
      Await_Lock( name_lock, read, name )
      Await_Lock( symtable_lock, exact )
```

```
      quick_list()
   end procedure ! exact_list

procedure symtab_boa () is
   ! The alternate handler for the BOA (beginning of action)
   ! event.  The initial states of the per-action variables for
   ! the current action are set to be the same as those of its
   ! immediate ancestor; the effect is that the child action may
   ! view modifications made by its parent action before the
   ! child action was created.

   status : action_status
   level  : action_level

   begin
      ! see if we're in a nested action
      Void( ActionManager @ Tell_ID( status, level ) )
      if level = nested_action then ! copy the parent's variables
         Self.inserted := ObjCopy( Parent.inserted )
         Self.deleted  := ObjCopy( Parent.deleted  )
      else ! this is a top-level action; allocate empty lists
         Self.inserted := new keyed_list
         Self.deleted  := new list
      end if
   end procedure ! symtab_boa

procedure symtab_commit () is
   ! The alternate handler for the commit action event.  If this
   ! is a nested action, we propagate the inserted and deleted
   ! lists of this action to its parent.

   status : action_status
   level  : action_level
```

```
   begin
      ! check whether we're in a nested action
      Void( ActionManager @ Tell_ID( status, level ) )
      if level = nested_action then
         Parent.inserted := ObjCopy( Self.inserted )
         Parent.deleted  := ObjCopy( Self.deleted  )
      end if
   end procedure ! symtab_commit

procedure symtab_top_precommit () is
   ! The alternate handler for the toplevel precommit action
   ! event.  We traverse the deleted and inserted lists for this
   ! action tree, performing the actual changes to the permanent
   ! symbol table.

   name      : name_type
   value     : value_type
   not_there : boolean
   n         : unsigned

   begin
      ! First, we will traverse the deleted list, and delete the
      ! given entries from the permanent symbol table
      n := 1
      loop
         name := Self.deleted @ nth( n, not_there )
         if not_there then
            exit .
         end if
         symtable[ hash( name ) ] @ remove( name )
         n += 1
      end loop

      ! Similarly, we traverse the inserted list for this action
```

```
      n := 1
      loop
         name := Self.inserted @ nth( n, value, not_there )
         if not_there then
            exit .
         end if
         symtable[ hash( name ) ] @ add( name, value )
         n += 1
      end loop
   end procedure ! symtab_top_precommit

inithandler is
   ! Here, we initialize the permanent symbol table.
   ! (Initialization of permanent structures is possible because
   ! the initialization handler of a recoverable object is
   ! performed implicitly as a toplevel precommit handler.)

   begin
      for i in hash_range loop  ! each bucket is initially empty
         symtable[ i ] := new bucket_list
      end loop
   end inithandler

end implementation.
```

C Resilient Symbol Table Implementation

This appendix contains the Aeolus source code for the implementation part of a resilient symbol table object.

```
implementation of object symtab
      !( name_type : type,  value_type : type )! is
   ! Single-copy symbol table object using the lock mechanisms of
   ! Aeolus/Clouds for synchronization and the critical region
   ! and shared constructs for mutual exclusion.  Since this
   ! object is not recoverable, we will explicitly release locks.

import keyed_list

!-----------------------------------------------------------------
! Each bucket of the hash table is a list of names and values,
! keyed by the name field.

type bucket_list is new keyed_list( name_type, value_type )

!-----------------------------------------------------------------
! The symbol table structure itself is an array of bucket lists.
! Each bucket is shared, and thus must be modified only within a
! critical region.

MAXBUCKET : const integer := 101     ! or whatever

type hash_range is new unsigned[ 1 .. MAXBUCKET ]

symtable : array[ hash_range ] of shared bucket_list

!-----------------------------------------------------------------
! The symtable_lock allows the entire symbol table to be
! locked.  This lock is set (in exact mode) in the
```

```
! exact_list operation for purposes of getting an exact
! listing of the state of the symbol table.  Operations which
! change the state of the symbol table must wait for completion
! of any outstanding exact_list operations and vice versa.

symtable_lock : lock ( exact    : [ exact    ] ,
                       nonexact : [ nonexact ] )

procedure hash ( name : name_type ) returns hash_range is
   ! This hash function is a local (nonpublic) procedure of
   ! the symtab object.

   begin
      NULL   ! the usual type of stuff
   end procedure ! hash

procedure insert (! name  : name_type    ,
                  ! value : value_type   ,
                  ! error : out boolean !) is
   ! The insert operation adds an entry to the appropriate
   ! bucket of the symbol table.

   dummy      : value_type
   bucket_num : hash_range

   begin
      bucket_num := hash( name )
      Await_Lock( symtable_lock, nonexact )
      region symtable[ bucket_num ] do
         error := symtable[ bucket_num ] @ find( name, dummy )
         if not error then
            symtable[ bucket_num ] @ add( name, value )
         end if
      end region
```

```
      Release_Lock( symtable_lock, nonexact )
   end procedure ! insert

procedure delete (! name  : name_type     ,
                  ! error : out boolean !) is
   ! If the delete operation finds an entry with value
   ! field = name in the appropriate bucket, it removes
   ! that entry.

   dummy      : value_type
   bucket_num : hash_range

   begin
      bucket_num := hash( name )
      Await_Lock( symtable_lock, nonexact )
      region symtable[ bucket_num ] do
         error :=
            not symtable[ bucket_num ] @ find( name, dummy )
         if not error then
            symtable[ bucket_num ] @ remove( name )
         end if
      end region
      Release_Lock( symtable_lock, nonexact )
   end procedure ! delete

procedure lookup (! name  : name_type     ,
                  ! error : out boolean !)
               ! returns value_type ! is
   ! The lookup operation sets a read lock on the
   ! name entry, and then tries to locate that entry with
   ! name field = name and returns its value if it
   ! succeeds.

   value      : value_type
```

```
    begin
       Await_Lock( symtable_lock, nonexact )
       error := not symtable[ hash( name ) ] @ find( name, value )
       Release_Lock( symtable_lock, nonexact )
       return value
    end procedure ! lookup

procedure quick_list() is
    ! The quick_list operation provides a quick (dirty)
    ! listing of names currently in the symbol table.

    begin
       for i in hash_range loop
          symtable[ i ] @ display()
       end loop
    end procedure ! quick_list

procedure exact_list () is
    ! The exact_list operation provides a listing of the
    ! exact state of the symbol table at a given point in time.
    ! To do this, it locks the whole symbol table, thereby
    ! excluding any changes during preparation of the listing.
    ! Thus, although exact_list, lookup, and
    ! quick_list operations may execute concurrently, and
    ! insert and delete operations which access
    ! different hash buckets may also execute concurrently,
    ! insert and delete operations must block on
    ! exact_list operations and vice versa.

    begin
       Await_Lock( symtable_lock, exact )
       quick_list()
       Release_Lock( symtable_lock, exact )
```

```
   end procedure ! exact_list

inithandler is
   ! Here, we initialize the symbol table.

   begin
      ! each bucket is initially empty
      for i in hash_range loop
         region symtable[ i ] do
            symtable[ i ] := new bucket_list
         end region
      end loop
   end inithandler

end implementation.
```

D Resilient Symbol Table with Resilient Type Construct

This appendix contains the Aeolus source code for the implementation part of a resilient symbol table object using the 'resilient type' construct.

```
implementation of object symtab
      !( name_type : type,  value_type : type )! is

! Single-copy symbol table object using the declarative resilient
! type feature to replace the imperative combination of the
! permanent and per-action variable features.

import keyed_list

! Each bucket of the hash table is a list of names and values,
! keyed by the name field.

type bucket_list is new keyed_list( name_type, value_type )

MAXBUCKET : const integer := 101     ! or whatever

type hash_range is new unsigned[ 1 .. MAXBUCKET ]

! The symbol table structure itself is an array of bucket lists.
! Here, the structure type is declared to be resilient, with a
! representation in permanent storage which is modifiable only at
! top-level precommit.  The resilient type specification also
! defines the relationship of the modifies operations of the
! object to the representation of the type.  The syntax used
! here is:

!      <operation name> (<key parameter> [, <value parameter>])
!          [reverse <operation name>] : <rep modification>
```

```
! The <rep modification> is a statement specifying the effect of
! the given operation on the representation of (a variable of)
! the resilient type.  If the operation may reverse the effect of
! another operation, this is indicated by use of the reverse
! clause.  The effect of the resilient type specification is, for
! each modifies operation, to generate an list which is used to
! maintain ''intentions'' of modifications caused by invocations
! of that operation by an action.  The ''intentions'' lists are
! automatically initialized for a new action and propagated up
! the action tree as in the symtab example using permanent and
! per-action variables.  Then, at top-level precommit, the
! ''intentions'' are translated automatically into modifications
! of the representation.  A visibility rule governing both the
! permanent representation and the modification ''intentions'' of
! an action is specified in the with visibility clause.  Finally,
! an iterator may be defined which yields all visible elements of
! the resilient type; thus, it may be specified to iterate over
! the ''intentions'' of an action as well as the permanent
! representation.

type symtable_type is
   resilient array[ hash_range ] of bucket_list
   with modifies operations
      insert ( name, value ) :
         rep[ hash( name ) ] @ add( name, value ) ,
      delete ( name ) reverse insert :
         rep[ hash( name ) ] @ remove( name )
      end operations
   visibility ( name : name_type, out value : value_type ) is
         insert( name, value )
      or (     not delete( name )
           and rep[ hash( name ) ] @ find( name, value ) )
      end visibility
   iterator ( out value : value_type ) returns name_type is insert :
```

```
         for i in bucket_range loop
            return rep[ i ] @ iterate( value )
         end loop
      end iterator
   end resilient

symtable : symtable_type

! The symtable_lock allows the entire symbol table to be locked.
! This lock is set (in exact mode) in the exact_list operation for
! purposes of getting an exact listing of the state of the symbol
! table.  Operations which change the state of the symbol table
! must wait for completion of any outstanding exact_list
! operations and vice versa.

symtable_lock : lock ( exact    : [ exact    ] ,
                       nonexact : [ nonexact ] )

! The NAME lock allows the user to lock the name which is to be
! used in one of the symbol table operations.  The purpose of
! this lock is to assure the view atomicity of these operations,
! that is, to provide synchronization such that concurrent users
! of the symbol table do not view the results of other actions
! before those actions are committed.

name_lock : lock ( write : []     ,
                   read  : [read] ) domain is name_type

procedure hash ( name : name_type ) returns hash_range is
   ! This hash function is a local (nonpublic) procedure of the
   ! symtab object.

   begin
      NULL   ! the usual type of stuff
   end procedure ! hash
```

```
procedure insert (! name  : name_type    ,
                  ! value : value_type   ,
                  ! error : out boolean !) is
   ! This operation invokes the insert operation of the resilient
   ! symtable to add the given item to the insertion
   ! ''intentions'' of the current action.

   dummy : value_type

   begin
      Await_Lock( name_lock, write, name )
      error := symtable @ visibility( name, dummy )
      if not error then
         Await_Lock( symtable_lock, nonexact )
         symtable @ insert( name, value )
      end if
   end procedure ! insert

procedure delete (! name  : name_type    ,
                  ! error : out boolean !) is
   ! This operation invokes the delete operation of the resilient
   ! symtable to add the given item to the deletion
   ! ''intentions'' of the current action.

   dummy : value_type

   begin
      error := FALSE
      Await_Lock( name_lock, write, name )
      if symtable @ visibility( name, dummy ) then
         Await_Lock( symtable_lock, nonexact )
         symtable @ delete( name )
      else
         ! name not in the permanent symbol table or inserted by
```

```
         ! this action
         error := TRUE
      end if
   end procedure ! delete

procedure lookup (! name  : name_type     ,
                  ! error : out boolean !) ! returns value_type ! is
   ! The lookup operation sets a read lock on the name entry, and
   ! then tries to locate that entry with name field = name and
   ! returns its value if it succeeds.

   value : value_type

   begin
      Await_Lock( name_lock, read, name )
      Await_Lock( symtable_lock, nonexact )
      error := not symtable @ visibility( name, value )
      return value
   end procedure ! lookup

procedure quick_list () is
   ! The quick_list operation provides a quick (dirty) listing of
   ! names currently in the symbol table by invoking the
   ! iterator of the resilient symtable.

   name  : name_type
   value : value_type

   begin
      for name in symtable @ iterate( value ) loop
         ! invoke display operations on name - value pair
      end loop
   end procedure ! quick_list

procedure exact_list () is
```

```
    ! The exact_list operation provides a listing of the exact
    ! state of the symbol table at a given point in time.  To do
    ! this, it locks the whole symbol table, thereby excluding any
    ! changes during preparation of the listing.  Thus, although
    ! exact_list, lookup, and quick_list operations may execute
    ! concurrently, and insert and delete operations which access
    ! different hash buckets may also execute concurrently, insert
    ! and delete operations must block on exact_list operations
    ! and vice versa.

    begin
       Await_Lock( name_lock, read, name )
       Await_Lock( symtable_lock, exact )
       quick_list()
    end procedure ! exact_list

end implementation.
```

E Ad-Hoc Replicated Symbol Table

This appendix contains the source code for an ad-hoc implementation of a replicated symbol table object. This implementation is based on the symbol table presented in earlier appendices. Is uses a master/slave (or 'hot spare') for replication control.

```
implementation of object symtab !( replica_number : integer,
                                   name_type      : type   ,
                                   value_type     : type   )! is

   ! A version of the resilient symbol table object presented in
   ! Appendix C using ad hoc techniques to implement
   ! replication.

import list, keyed_list

! Here, we give the names of alternate handlers for some of the
! action events.  Note that we need not override the abort event.

action events
             symtab_boa overrides boa,
          symtab_commit overrides commit,
   symtab_top_precommit overrides toplevel_precommit

! The per-action variables for the symbol table are where we
! maintain the ''intention lists'' of entries inserted and deleted
! by an action.  The inserted list entries are keyed on the
! name field, but also include the value field.  The deleted
! list entries need merely give the name field.  These variables
! are initialized in the handler for the BOA action event.

per action
   inserted : keyed_list( name_type, value_type )
   deleted  : list( name_type )
end per action
```

```
! The definitions of MAXREPLICAS, replica_range, and
! version_vector should actually appear in the definition part of
! symtab, but are shown here for convenience.

MAXREPLICAS : const integer := 2

constraint replica_range is integer[ 1 .. MAXREPLICA ]

type version_vector is array[ replica_range ] of integer

!
! The actual declarations of the implementation part.
!

recoverable
   here, there : replica_range
      ! for storing values of replica numbers

   partner : symtab()
      ! Object pointer to the partner object

   master : boolean
      ! remember whether this instance is master or slave

   local_version : version_vector
      ! The local_version vector is used to store version numbers
      ! of the local state (the here entry) and of the last
      ! version of the local state known to be consistent with
      ! the partner's state (the there entry).  Note, however,
      ! that only one copy (per instance) of the actual state is
      ! maintained.
end recoverable

! Each bucket of the hash table is a list of names and values,
```

```
! keyed by the name field.  The list objects are kept in
! permanent storage, and thus modify operations on them may be
! invoked only during toplevel precommit.  (However, examine
! operations may be invoked at any time.)

type bucket_list is permanent new keyed_list( name_type, value_type )

! The symbol table structure itself is an array of bucket lists.
! The array is also kept in permanent storage, and may be altered
! only at toplevel precommit.  Since action management ensures
! that only one action may be in the toplevel precommit handler
! at a time, there is no need to explicitly enforce mutual
! exclusion on the symbol table buckets, as is done in the
! nonrecoverable version of the symtab object by means of
! critical regions.

MAXBUCKET : const integer := 101     ! or whatever

type hash_range is new unsigned[ 1 .. MAXBUCKET ]

symtable : permanent array[ hash_range ] of bucket_list

! The symtable_lock allows the entire symbol table to be locked.
! This lock is set (in exact mode) in the exact_list operation for
! purposes of getting an exact listing of the state of the symbol
! table.  Operations which change the state of the symbol table
! must wait for completion of any outstanding exact_list
! operations and vice versa.

symtable_lock : lock ( exact    : [ exact    ] ,
                       nonexact : [ nonexact ] )

! The NAME lock allows the user to lock the name which is to be
! used in one of the symbol table operations.  The purpose of
```

```
! this lock is to assure the view atomicity of these operations,
! that is, to provide synchronization such that concurrent users
! of the symbol table do not view the results of other actions
! before those actions are committed.

name_lock : lock ( write : []     ,
                   read  : [read] ) domain is name_type

procedure hash ( name : name_type ) returns hash_range is
   ! This hash function is a local (nonpublic) procedure of the
   ! symtab object.

   begin
      NULL   ! the usual type of stuff
   end procedure ! hash

procedure sym_find ( name  : name_type      ,
                     value : out value_type ) returns boolean is
   ! The sym_find routine is a local (nonpublic) procedure of the
   ! symtab object.  It assumes that the caller has obtained the
   ! necessary locks.

   begin
      return    Self.inserted @ find( name, value )
             or (     not Self.deleted @ find( name )
                  and symtable[ hash( name ) ] @ find( name, value ) )
   end procedure ! sym_find

procedure sym_insert (! name       : name_type    ,
                      ! value      : value_type   ,
                      ! newversion : integer      ,
                      ! error      : out boolean !) is
   ! The sym_insert operation adds an entry to the inserted list
   ! for this action, if the entry is not found; otherwise, error
```

```
    ! is set to TRUE.  The entry will placed into the permanent
    ! symbol table at top-level precommit.
    ! The sym_insert operation is called either by the insert
    ! operation shown below (if this instance is the master), or
    ! by the partner as an update operation (if this instance is
    ! the slave).  The symtable_lock and name_lock are obtained by
    ! the caller.  If newversion is greater than 0 (that is, the
    ! sym_insert operation was called remotely), then this new
    ! version number is installed in the local_version array.

    dummy : value_type

    begin
       error := sym_find( name, dummy )
       if not error then
          Self.inserted @ add( name, value )
       end if
    end procedure ! sym_insert

procedure insert (! name  : name_type    ,
                  ! value : value_type   ,
                  ! error : out boolean !) is
    ! If this instance is the master, it sets the symtable_lock
    ! and the name_lock, and then calls the sym_insert operation
    ! both locally and at the slave partner (if available) to do
    ! the actual insertion.  If this instance is the slave, it
    ! calls the master (if available) to do the insertion just as
    ! if the call had originated there.  If the master is not
    ! available, the slave makes itself the master and performs
    ! the insertion.  Note that, since only the master can call
    ! the sym_insert operation (which actually performs the
    ! insertion), it is not necessary to set locks at the slave.

    newversion : integer
```

```
aid        : action_id
success    : boolean

begin
   if master then
      newversion := local_version[ here ] + 1
      AwaitLock( symtable_lock, nonexact )
      AwaitLock( name_lock, write, name )
      sym_insert( name, value, 0, error )
      local_version[ here ] := newversion

      if partner_available then ! update at the partner
         aid := action( partner @ sym_insert( name, value,
                                    newversion, error ) )
         Wait_Completion( aid, success )
            ! block until the nested action commits or aborts
         if Status( aid, FALSE ) = committed then
            local_version[ there ] := newversion
         else
            partner_available := FALSE
         end if
      end if

   else
      ! this instance is the slave; request the master
      ! to update both of us
      aid := action( partner @ insert( name, value, error ) )
      Wait_Completion( aid, success )
      if Status( aid, FALSE ) = committed then ! become master
         partner_available := FALSE
         master            := TRUE
         newversion        := local_version[ here ] + 1
         AwaitLock( symtable_lock, nonexact )
         AwaitLock( name_lock, write, name )
```

```
            sym_insert( name, value, 0, error )
            local_version[ here ] := newversion
         end if
      end if
   end procedure ! insert

procedure symtab_boa () is
   ! The alternate handler for the BOA (beginning of action)
   ! event.  The initial states of the per-action variables for
   ! the current action are set to be the same as those of its
   ! immediate ancestor; the effect is that the child action may
   ! view modifications made by its parent action before the
   ! child action was created.

   status : action_status
   level  : action_level

   begin
      ! see if we're in a nested action
      Void( ActionManager @ Tell_ID( status, level ) )
      if level = nested_action then ! copy the parent's variables
         Self.inserted := ObjCopy( Parent.inserted )
         Self.deleted  := ObjCopy( Parent.deleted  )
      else ! this is a top-level action; allocate empty lists
         Self.inserted := new keyed_list
         Self.deleted  := new list
      end if
   end procedure ! symtab_boa

procedure symtab_commit () is
   ! The alternate handler for the commit action event.  If this
   ! is a nested action, we propagate the inserted and deleted
   ! lists of this action to its parent.
```

```
    status : action_status
    level  : action_level

    begin
       ! check whether we're in a nested action
       Void( ActionManager @ Tell_ID( status, level ) )
       if level = nested_action then
          Parent.inserted := ObjCopy( Self.inserted )
          Parent.deleted  := ObjCopy( Self.deleted  )
       end if
    end procedure ! symtab_commit

procedure symtab_top_precommit () is
    ! The alternate handler for the toplevel precommit action
    ! event.  We traverse the deleted and inserted lists for this
    ! action tree, performing the actual changes to the permanent
    ! symbol table.

    name      : name_type
    value     : value_type
    not_there : boolean
    n         : unsigned

    begin
       ! First, we will traverse the deleted list, and delete the
       ! given entries from the permanent symbol table
       n := 1
       loop
          name := Self.deleted @ nth( n, not_there )
          if not_there then
             exit .
          end if
          symtable[ hash( name ) ] @ remove( name )
          n += 1
```

```
      end loop

      ! Similarly, we traverse the inserted list for this action
      n := 1
      loop
         name := Self.inserted @ nth( n, value, not_there )
         if not_there then
            exit .
         end if
         symtable[ hash( name ) ] @ add( name )
         n += 1
      end loop
   end procedure ! symtab_top_precommit

procedure set_partner
      (! p : symtab(),  rep_number : replica_range !) is
   ! The set_partner operation is used by the creating process to
   ! initialize the partner capability.

   begin
      partner := p
      there   := rep_number
   end procedure ! set_partner

inithandler is
   ! Here, we initialize the permanent symbol table.
   ! (Initialization of permanent structures is possible because
   ! the initialization handler of a recoverable object is
   ! performed implicitly as a top-level precommit handler.)

   begin
      here := replica_number
      master := here = 1    ! arbitrary choice
      local_version := version_vector"{ 0 : MAXREPLICAS }
```

```
      for i in hash_range loop  ! each bucket is initially empty
         symtable[ i ] := new bucket_list
      end loop
   end inithandler

end implementation.
```

F Aeolus Distributed Locking Primitives

This appendix contains the Aeolus definition part that serves as the user interface to the Distributed Locking primitives.

```
definition of pseudo object DistLock is

   ! Interfaces to primitives provided for support of the
   ! Distributed Locking mechanism.  This pseudo-object is
   ! imported automatically by every availspec, and is not
   ! available for use by other compilands.

type replica_number is new unsigned
   ! A replica_number is used to name an individual replica of a
   ! group.  The naming scheme used here is the ''horizontal''
   ! method as described in Chapter VII of this dissertation.
   ! The replica_number is concatenated by the system to the
   ! capability of the object to which the invoking availspec
   ! belongs to form an extended capability as defined by the
   ! horizontal scheme.

type version_number is new longuns
   ! A version_number is used to compare the currency of
   ! the states of replicas.  The version number of an object is
   ! incremented whenever an invocation is performed on it, or
   ! when the state of the objected is updated by use of one of
   ! the designated operations described below.

operations

   procedure lock_replica ( rep : replica_number      ,
                            ver : out version_number )
            returns boolean modifies
      ! The lock_replica operation obtains the
```

```
    ! currently-requested lock at the replica denoted by rep.
    ! This operation should be invoked only within a lock event
    ! handler.  The lock variable, domain value, and mode
    ! requested are obtained from the context of the lock
    ! event which caused the invocation of the handler.
    ! The replica denoted by rep is added to a list of the
    ! replicas touched by the current action.
    ! The version number of the state of rep is returned
    ! in the out parameter ver.
    ! If lock_replica is unable to obtain the lock on
    ! rep, or if the requested lock is already held
    ! at rep by the current action, the operation returns
    ! FALSE, otherwise TRUE.

procedure invoke_replica ( rep : replica_number )
          returns boolean modifies
    ! The invoke_replica operation causes the current operation
    ! to be executed at the replica denoted by rep.  This
    ! operation should be invoked only within a copy event
    ! handler.  The operation number and other parameters are
    ! obtained from the context of the lock which caused the
    ! invocation of the handler.  The version number of rep
    ! is set to the value of that of the invoking object.
    ! This operation is used for implementing state copying by
    ! idemexecution.  If the invocation on rep is
    ! unsuccessful, the operation returns FALSE, otherwise
    ! TRUE.

procedure broadcast_shadows () returns boolean modifies
    ! The broadcast_shadow operation causes the ''shadow set''
    ! of the permanent state of the current action to be
    ! broadcast to all replicas at which locks were obtained by
    ! the current action via the lock_replica operation.
    ! The version numbers of the locked replicas are updated
```

```
    ! to equal that of the invoking object.  This
    ! operation should be invoked only within a copy event
    ! handler.  This operation is used for implementing state
    ! copying by cloning using shadows.  If all locked
    ! replicas successfully receive the shadow set, the
    ! operation returns TRUE, otherwise FALSE.

procedure get_state ( rep : replica_number )
          returns boolean modifies
    ! The get_state operation causes the state of the
    ! replica denoted by rep to be transmitted to the
    ! current object.  The state is installed at the current
    ! object, and its version number set to that of rep.
    ! If the transmission or installation fails, the operation
    ! returns FALSE, otherwise TRUE.

procedure send_state ( rep : replica_number )
          returns boolean modifies
    ! The send_state operation causes the state of the
    ! current object to be transmitted to the replica denoted
    ! by rep.  The state is installed at rep, and
    ! its version number set to that of the current object.  If
    ! the transmission or installation fails, the operation
    ! returns FALSE, otherwise TRUE.

procedure invoke_acceptor ( rep   : replica_number ,
                            state : address        ,
                            len   : longuns        ) modifies
    ! The invoke_acceptor operation causes the invocation
    ! of the accept event handler at the replica denoted by
    ! rep.  The information the address of which is given by
    ! state and which is of length len bytes is copied to the
    ! environment of the accept handler at rep.  This operation
    ! may be used in a copy event handler to implement state
```

```
        ! copying by cloning using logs, or in a reinit event
        ! handler to implement state reconciliation strategies.

    procedure degree () returns replica_number examines
        ! The degree operation returns the total number of
        ! replicas of the current object including itself.

    procedure my_replica () returns replica_number examines
        ! The my_replica operation returns the replica number of
        ! the current object.

    procedure my_version () returns version_number examines
        ! The my_version operation returns the version number of
        ! the current object's state.

    procedure quorum_size () returns replica_number examines
        ! The quorum_size operation returns the minimum size of
        ! a quorum for the currently-requested lock mode.

    procedure currently_locked () returns replica_number
        ! The currently_locked operation returns the number of
        ! replicas on which the currently-requested lock mode has
        ! been obtained, including the current object.

end definition. ! DistLock
```

G Clouds Action Manager Interface

This appendix contains the interface to the Clouds action manager.

```
definition of pseudo object ActionManager is

   ! This object is implicitly imported by every object of class
   ! RECOVERABLE or AUTORECOVERABLE, and an instance with the
   ! same name is implicitly declared.  Operations on this object
   ! instance need not be qualified by the object instance name.

   type action_status is ( active    , quiesced , precommitted ,
                           committed , aborted  , not_action   )
      ! The meanings of the action states are as follows:
      !     active       - the action is still running
      !     quiesced     - the action has not committed, but
      !                    may not perform any further operations
      !     precommitted - the action has successfully prepared
      !                    for commit
      !     committed    - the action has successfully completed
      !     aborted      - the action has been aborted
      !     not_action   - the action ID referenced is unknown
      !                    by the system

   type action_level is ( toplevel_action, nested_action )

   type action_ID is private
      ! Type defined by kernel object.  (This actually is what is
      ! called the activity_ID type in [Ken186].)

   type timeout_type is ( default, relative, absolute )
      ! The value of a timeout_type indicates the meaning of an
      ! associated timeout_value, as follows:
      !     If default, then the system-defined default timeout
```

```
   !        value is used, and the supplied timeout value is
   !        ignored.
   !     If absolute, then the supplied timeout value is
   !        interpreted as the time the action will cease to
   !        exist in Lamport real time (i.e., close to but not
   !        necessarily equal to wall-clock time.)
   !     If relative, then the supplied timeout value is
   !        interpreted as a ''displacement'' from the current
   !        Lamport global time.

type timeout_value is new longuns

operations

   procedure Tell_ID ( status : out action_status ,
                       level  : out action_level  )
                    returns action_ID examines
      ! If the caller is attached to an action, returns the
      ! action ID, and places the status and nesting level of
      ! the caller's action in the \f6out parameters.  If the
      ! caller is not attached to an action, the parameter
      ! status has value not_action.

   procedure Status ( aid   : in action_ID ,
                      clear : in boolean   )
                   returns action_status examines
      ! Returns the status of the action referenced by aid.
      ! If the parameter clear is TRUE, and the action is in
      ! the committed or aborted state, all information about
      ! the action is forgotten by the system.

   procedure Set_Timeout ( aid    : in action_ID      ,
                           type   : in timeout_type   ,
                           value  : in timeout_value  ,
```

```
                        status : out action_status ) modifies
   ! If the parameter aid refers to a valid action, and the
   ! status of the action is active, quiescent, or
   ! precommitted, the timeout value previously assigned to
   ! that action is replaced by the specified value (in
   ! milliseconds).  In any case, the status of the action
   ! is returned in the \f6out parameter status.  See above
   ! for explanation of timeout_type.

procedure Get_Timeout ( aid    : in action_ID      ,
                        status : out action_status )
                      returns timeout_value examines
   ! If the parameter aid refers to a valid action, and the
   ! status of the action is active, quiescent, or
   ! precommitted, the timeout value previously assigned to
   ! that action (in milliseconds) is returned.  This
   ! timeout value is the absolute Lamport clock time when
   ! this action will expire (see explanation of
   ! timeout_type above).  If the above conditions are not
   ! fulfilled, value is zero.  The status of the action is
   ! returned in the \f6out parameter status.

procedure Get_Time () returns timeout_value examines
   ! The current global Lamport clock time is returned.

procedure Abort ( aid : in action_ID )
                returns action_status modifies
   ! Abort the action designated by aid, returning status
   ! aborted if successful.  If unsuccessful, the current
   ! status of that action is returned.

procedure Abort_Myself () returns action_status modifies
   ! Abort the current action, returning status aborted if
   ! successful.  If unsuccessful, the current status of
```

```
        ! that action is returned.
        ! [Same as Abort( Tell_ID(...) )].

    procedure Commit () returns action_status modifies
        ! Commit will be interpreted by action management
        ! dependent on the current context of the calling
        ! sequence.  If the current process is not linked to an
        ! action, it will always be interpreted as an object
        ! management return.  If the process is attached to an
        ! action, then it will be interpreted as an action
        ! management Commit operation only if it is the initial
        ! operation in the calling sequence.  This is somewhat
        ! equivalent to the statement ''Object operations
        ! performed during an action must totally finish.''

    procedure Wait_Completion ( aid     : action_ID   ,
                                success : out boolean ) examines
        ! This is a very general wait.  It assumes very little
        ! about the relationship between the current (calling)
        ! action and the action denoted by aid, i.e., whether
        ! aid is a child of the current action.  Wait_Completion
        ! sets the parameter success to TRUE if the wait was
        ! successful, or FALSE if the action denoted by aid is
        ! unknown.  The caller is expected to obtain the actual
        ! status of aid (and, if aid is a child of the current
        ! action, clear its child status field) with an
        ! Action_Status call.

end definition. ! ActionManager
```

H Aeolus/Clouds Feature Summary

In this appendix, a summary of the features provided by the Aeolus/Clouds system is presented. In particular, the portion of the system which supports each feature is mentioned. The features are divided into three main groups: those which support objects in general, those which support actions, and those which support the interactions of actions with objects.

In this discussion, the terms "kernel" and "object manager" (OM), "storage manager" (SM), and "action manager" (AM) refer to the prototypes as described by Spafford [Spaf86], Pitts [Pitt86], and Kenley [Kenl86], respectively.

H.1 Features Supporting Objects

Specification of the state and operations of an object in Aeolus is achieved by the combination of a `definition` part and an `implementation` part for that object. The `definition` part specifies the external interface of the object; the `implementation` part provides specifications of the object data as well as code for the operations and any internal procedures of the object.

H.1.1 Persistent State

The persistence of the state of a Clouds object between successive invocations of that object is guaranteed by the Clouds object management system. The object state consists of the global variables of the object as specified in the object implementation part; local variables of operations and internal procedures are maintained on a *per-process stack* and are thus considered *volatile* (*i.e.*, they do not persist between invocations). The Aeolus compiler provides the OM with information about the size and location of the object state via the object header. Note that *persistence* does not imply *permanence*, which is a resilience property (see the discussion of *permanent variables* below).

H.1.2 Object Instance Creation

Object instance creation is achieved in Aeolus by use of an *allocator*, which generates a capability to an object of the specified type. This capability may then be assigned to a variable or used for invocations. Use of the allocator construct to create a Clouds object instance requires the compiler to generate an invocation of the `Create` operation on the `TypeTemplate` of the specified object type. The `TypeTemplate` may in turn generate kernel calls as required for creating and initializing the various segments required by the object state.

If an allocator is used to create an instance of a non-Clouds object, the compiler generates a call to the memory allocation routine of the Aeolus RTS, which creates a data area of the appropriate size on the heap for use as the object state. The pointer thus generated is used as a "sysname" for the object.

H.1.3 Object Invocation

Invocation of an operation on a Clouds object currently requires three steps: reformatting of the `in` parameters of the invocation from the internal Aeolus format as provided on the object stack into the format required for Clouds RPC; a kernel call to perform the actual RPC (called from the Aeolus RTS routine `ObjInvoke`); and reformatting of the `out` parameters of the invocation from the Clouds kernel format into the internal Aeolus format. The reformatting operations are artifacts of the kernel parameter format, and are performed by the common object operation entry point (`ObjEntry`) provided by the Aeolus runtime system. (A more congenial design of the kernel parameter format would eliminate the need for this reformatting.)

The RPC kernel call is provided with the addresses of the newly-formatted `in` and `out` parameter areas, as well as the number of the object operation to be invoked and the invoker's access rights to the invokee. (A description of operation invocation from the kernel side, as well as the necessary bookkeeping by the kernel, is provided in Spafford's dissertation, p. 43ff and p. 82ff.) The kernel takes care of copying the parameters to the remote node in the case of a remote invocation, and passes its parameters to the `ObjEntry` routine of the invokee, the address of which it obtains from the object header. `ObjEntry` has access to the vector of operation addresses in the invoked object, and calls the operation

denoted by the given operation number. The operation returns to the `ObjEntry` routine, which invokes the kernel to handle return of the RPC.

Invocation of an operation on a non-Clouds object is essentially the same as a regular internal procedure call, with the exception that the pointer to the object state (on the heap) is passed as a "hidden" parameter to the operation.

H.1.4 Object Event Handlers

The object events include at least the initialization and deletion of an object. The code of user-specified handlers for object events is specified in the implementation part of an object. The addresses of object event handlers are placed at predefined indices in the operation address vector for that object. If a handler for one of the object events is not provided by the user, a default handler for that event is generated by the Aeolus compiler (possibly using code in the RTS). In the prototype, the handlers are invoked by the kernel calls for object creation and deletion, although equivalently these could be called by the `Create` and `Delete` operations of the object type's `TypeTemplate`.

H.2 Features Supporting Actions

Support for actions consists of the action invocation mechanism as well as several additional operations provided by the AM. The use of an action invocation causes the compiler to generate a call to the `Create_Action` operation of the AM. The interface to this operation is described on p. 18 of Kenley's thesis. (This operation is not made available in the programmer's interface to the AM.) The interface requires specification of whether creation of a top-level or nested action is desired; if a non-action process requests a nested action, a top-level action is created.

The `Create_Action` operation returns an *action identifier* which may be used in invocations on the AM to query and modify the status of the new action. The AM operations available to the programmer are described in Appendix G.

H.3 Features Supporting Action/Object Interactions

Support for interactions of actions with objects includes: features for specifying mutual exclusion; features for specifying view atomicity via synchronization; and features supporting resilience, including action event handlers, recoverable areas, per-action variables, and permanent variables.

H.3.1 Mutual Exclusion: Critical Regions

When a type is given the attribute **shared**, the compiler associates a semaphore with each variable or structure element of that type. The use of a critical region causes the Aeolus compiler to generate `P` and `V` calls on the semaphore associated with the shared variable specified in the critical region; these calls encapsulate the critical region, ensuring exclusive access to the region. Operations on semaphores are supported by the kernel.

H.3.2 Synchronization: Locks and Autosynch

The specification of a lock type causes the Aeolus compiler to generate a matrix of mode compatibilities for that type based on the given compatibility clause. This matrix is stored with the object state. Each row of the matrix corresponds to one of the modes of the lock, and specifies which modes of the lock are compatible with that mode.

Information about the locks held by an action is kept with the action control block for that action. This information is kept outside of the object state for efficiency reasons, *i.e.*, to avoid having to map in the object state for each object touched by an action in order to propagate the locks held by that action to its parent. The kernel structures necessary for this bookkeeping are described on p. 42-47 of Kenley's thesis. A parallel structure is maintained in the software PCB for a non-action process to hold information about locks obtained by that process.

If a lock is specified to have a **domain**, any code necessary to interpret domain values in calls on that lock is generated by the Aeolus compiler. The lock operations supported by the AM treat domain values passed to them as uninterpreted bit strings, as described on p. 41 of Kenley's thesis.

If an object is given the attribute **autosynch**, the compiler generates a domainless

multiple reader/single writer lock. (Recall that a domainless lock effictively has a domain of the entire object.) Also, at the beginning of each operation with the attribute `modify`, the lock is set in **write** mode; at the beginning of each **examine** operation, the lock is set in **read** mode. If the object is a non-Clouds or **nonrecoverable** Clouds object, code is generated at the end of the operation to release the lock. If an operation has neither of the attributes **modify** or **examine**, the compiler does not generate code to acquire or release the lock.

H.3.3 Action Event Handlers

The compiler generates action event handlers in essentially the same fashion as object event handlers, including the use of default handlers from the RTS when no user-specified handler is provided. Invocation of action event handlers, however, is performed by the AM. The AM may be requested to invoke an action event by three means: an explicit request by an object touched by the action; an implicit request, such as a return from an operation invoked as an action (which causes a return to the AM itself, and is considered an implicit commit); or an explicit request by an external process or action. Additional AM support for action event handling is described on p. 63-77 of Kenley's thesis.

H.3.4 Recoverable Areas

Information on the size of each recoverable area of an object is stored by the Aeolus compiler in that object's header. This information is used by the **Create** operation of the **TypeTemplate** for that object type to generate segment information in the kernel object descriptor for the new instance. The AM also uses this information to determine when it must call on the SM for special treatment.

As described on p. 35-39 of Kenley's thesis, when an action enters an object with a recoverable area (RA) for the first time, the AM copies the page table descriptors for the RA from the action's parent (or, for a top-level action, from the object's primary page map). This copy of the page table descriptors forms the action's *version* of the RA, and is used on subsequent invocations of the object by that action. Support required from the AM to manage the RA on operation return is described on p. 64 of Kenley's thesis. On commit of a nested action, the AM replaces the parent action's version of the RA with that

of the committing action; on top-level commit, the AM replaces the appropriate portion of the primary page map with the committed version (Kenley, p. 74). On abort, the current action's version is cleaned up (Kenley, p. 77).

H.3.5 Autorecoverable Objects

When an object is given the attribute **autorecoverable**, the Aeolus compiler essentially considers its entire state to be a recoverable area. Management of this RA is identical to that of user-specified RAs. Also, the compiler generates default handlers for all action events in an `autorecoverable` object.

H.3.6 Per-Action Variables

Treatment of per-action variables is similar in many respects to that of recoverable areas. The Aeolus compiler places information about the size of the per-action variables of an object in the object's header. This information is used by the AM to generate space for a new copy of the per-action variables when an action first touches the object.

Because the action must access both its own per-action variables and those of its parent, the per-action variables are not made part of the object page map for this action (as are recoverable areas). Rather, the per-action variables are maintained by the AM, and pointers for those of the current action and for those of its parent are passed as "hidden" parameters on each operation invocation (see p. 63 of Kenley's thesis for a rationale for this design).

All other maintenance of per-action variables, including propagation up the action tree, is the responsibility of the object programmer. This may be accomplished by means of programmer-specified action event handlers.

H.3.7 Permanent Variables

When a type is given the attribute **permanent**, the Aeolus compiler allocates space for variables or structures of that type in *permanent storage.* (Dynamically-allocated values of that type are allocated in the *permanent heap.*) These values may be initialized during the initialization object event; otherwise, they may be modified only during the top-level precommit action event. At all other times, they are considered to be read-only values.

The compiler places information about the location and extent of the permanent storage area in the object's header. This information is used by the `Create` operation of the `TypeTemplate` of that object type to generate segment information in the kernel object descriptor for each new instance. It is also used by the AM to signal the SM that this segment requires special treatment during top-level precommit or abort, as described on p. 69-77 of Kenley's thesis. During top-level precommit, the SM ensures through the use of *shadows* that modifications to permanent storage are atomic.

References

[Aham87] Ahamad, M., P. Dasgupta, R. J. LeBlanc, and C. T. Wilkes. 'Fault-Tolerant Computing in Object Based Distributed Operating Systems.' *Proceedings Of The Sixth Symposium On Reliability in Distributed Software and Database Systems*, (IEEE Computer Society), Williamsburg, VA, March 1987. Technical Report GIT-ICS-87-16, School of Information and Computer Science, Georgia Insitute of Technology, Atlanta, GA, 1987.

[Aham87a] Ahamad, M., and P. Dasgupta. 'Parallel Execution Threads: An Approach to Fault-Tolerant Actions.'

[Allc82] Allchin, J. E., and M. S. McKendry. 'Object- Based Synchronization and Recovery.' Technical Report GIT-ICS-82-15, School of Information and Computer Science, Georgia Insitute of Technology, Atlanta, GA, 1982.

[Allc83] Allchin, J. E., and M. S. McKendry. 'Synchronization and Recovery of Actions.' *Proceedings Of The Second Symposium On Principles Distributed Computing* (ACM SIGACT/SIGOPS), Montreal, August 1983.

[Allc83a] Allchin, J. E. 'An Architecture for Reliable Decentralized Systems.' Ph.D. Diss., School of Information and Computer Science, Georgia Insitute of Technology, Atlanta, GA, 1983.

[Birm84] Birman, K. P., T. A. Joseph, T. Raeuchle, and A. El-Abbadi. 'Implementing Fault-Tolerant Distributed Objects.' *Proceedings Of The Fourth Symposium On Reliability in Distributed Software and Database Systems*, Silver Spring, MD, October 1984.

[Birm85] Birman, K. P. 'Replication and Fault-Tolerance in the ISIS System.' *Proceedings Of The Tenth Symposium On Operating System Principles* (ACM SIGOPS), Orcas Island, Washington, December 1985.

[Birm87] Birman, K. P., and T. A. Joseph. 'Exploiting Virtual Synchrony in Distributed Systems.' Technical Report TR 87-811, Department of Computer Science, Cornell University, Ithaca, NY, February 1987.

[Dasg87] Dasgupta, P., R. LeBlanc, and W. Appelbe. 'The Clouds Distributed Operating System: Functional Description, Implementation Details, and Related Work.' Technical

Report GIT-ICS-87-28, School of Information and Computer Science, Georgia Insitute of Technology, Atlanta, GA, 1987.

[El-A85] El-Abbadi, A., D. Skeen, and F. Cristian. 'An Efficient, Fault-Tolerant Protocol for Replicated Data Management.' *Proceedings Of The Fourth Symposium On Principles Of Database Systems*, (ACM SIGACT/SIGMOD), March 1985.

[Grei86] Greif, I., R. Seliger, and W. Weihl. 'Atomic Data Abstractions in a Distributed Collaborative Editing System.' *Conference Record Of The Thirteenth Symposium On Principles Of Programming Languages* (ACM SIGACT/SIGPLAN), St. Petersburg Beach, FL, January 1986.

[Herl84] Herlihy, M. 'Replication Methods for Abstract Data Types.' Ph.D. Diss., Laboratory for Computer Science, Massachusettes Institute of Technology, Cambridge, MA, May 1984.

[Herl85] Herlihy, M. 'Atomicity vs. Availability: Concurrency Control for Replicated Data.' Technical Report CMU-CS-85-108, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, February 1985.

[Herl85a] Herlihy, M. 'Using Type Information to Enhance the Availability of Partitioned Data.' Technical Report CMU-CS-85-119, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, April 1985.

[Herl87] Herlihy, M., and J. M. Wing. 'Avalon: Language Support for Reliable Distributed Systems.' *Proceedings Of The Seventeenth International Symposium On Fault-Tolerant Computing*, Pittsburgh, PA, July 1987.

[Hone86] Honeywell, Inc. 'Fault Tolerant Distributed Systems.' Interim Scientific Report, Computer Science Center, Honeywell, Inc., Golden Valley, MN, November 1986.

[Jose85] Joseph, T. A. 'Low-Cost Management of Replicated Data.' Ph.D. Diss., Department of Computer Science, Cornell University, Ithaca, NY, November 1985.

[Jose86] Joseph, T. A., and K. P. Birman. 'Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems.' *Transactions On Computer Systems* (ACM) 4, no. 1, February 1986.

[Kenl86] Kenley, G. G. 'An Action Management System for a Distributed Operating System.' M.S. Thesis, School of Information and Computer Science, Georgia Insitute of Technology, Atlanta, GA, 1986.

[LeBl85] LeBlanc, R. J., and C. T. Wilkes. 'Systems Programming with Objects and Actions.', *Proceedings Of The Fifth International Conference On Distributed Computing Systems*, Denver, CO, July 1985.

[Lisk83] Liskov, B., M. Herlihy, P. Johnson, G. Leavens, R. Scheifler, and W. Weihl. 'Preliminary Argus Reference Manual.' *Programming Methodology Group Memo 39*, Laboratory for Computer Science, Massachussetts Institute of Technology, Cambridge, MA, October 1983.

[Lisk83a] Liskov, B., and R. Scheifler. 'Guardians and Actions: Linguistic Support for Robust Distributed Systems.' *Transactions On Programming Langauges And Systems* (ACM) 5, no. 3, July 1983.

[Lisk84] Liskov, B. 'Overview of the Argus Language and System.' *Programming Methodology Group Memo 40*, Laboratory for Computer Science, Massachussetts Institute of Technology, Cambridge, MA, February 1984.

[Long87] Long, D. D. E., and J.-F. Paris. 'On Improving the Availability of Replicated Files.' *Proceedings Of The Sixth Symposium On Reliability in Distributed Software and Database Systems*, (IEEE Computer Society), Williamsburg, VA, March 1987.

[McKe85] McKendry, M. S. 'Ordering Actions for Visibility.' *Transactions On Software Engineering* (IEEE) 11, no. 6, June 1985.

[Pitt86] Pitts, D. V. 'Storage Management for a Reliable Decentralized Operating System.' Ph.D. Diss., School of Information and Computer Science, Georgia Insitute of Technology, Atlanta, GA, 1986.

[Pitt87] Pitts, D. V., and P. Dasgupta. 'Object Memory and Storage Management in the Clouds Kernel.' Technical Report GIT-ICS-87-15, School of Information and Computer Science, Georgia Insitute of Technology, Atlanta, GA, 1987.

[Skee85] Skeen, D. 'Determining the Last Process to Fail.' *Transactions On Computer Systems* (ACM) 3, no. 1, February 1985.

[Spaf86] Spafford, E. H. 'Kernel Structures for a Distributed Operating System.' Ph.D. Diss., School of Information and Computer Science, Georgia Insitute of Technology, Atlanta, GA, 1986.

[Weih83] Weihl, W., and B. Liskov. 'Specification and Implementation of Resilient Atomic Data Types.' *Symposium On Programming Language Issues In Software Systems*, June 1983.

[Wilk85] Wilkes, C. T. 'Preliminary Aeolus Reference Manual.', Technical Report GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985.

[Wilk86] Wilkes, C. T., and R. J. LeBlanc. 'Rationale for the Design of Aelous: A Systems Programming Language for an Action/Object System.', *Proceedings Of The 1986 International Conference On Computer Langauges* (IEEE Computer Society), Miami, FL, October 1986.

[Wilk87] Wilkes, C. T. 'Programming Methodologies for Resilience and Availability.' Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987.

Fault Tolerant Software Technology for Distributed Computer Systems

Final Report

August 4, 1988

F30602-86-C-0032
G36-645

Richard J. LeBlanc

1. Summary of Project

This report documents the results of the project entitled "Fault Tolerant Software Technology for Distributed Computing Systems," a two year effort performed at Georgia Institute of Technology as part of the Clouds Project. The Clouds Project is building an object-oriented distributed operating system and studying how such a system supports the development of distributed applications, with a particular concern for highly available, fault-tolerant applications. The Clouds kernel supports *objects* as the fundamental encapsulation of data. Objects define permanent virtual address spaces and may allow access to and modification of their data through arbitrary, programmer-defined operations. Objects operations are invoked using capabilities which allow system-wide access to an object via the kernel-based capability interpretation mechanism. The kernel also provides *atomic actions* (corresponding roughly to the database notion of atomic transactions) in order to support the construction of reliable applications.

The design philosophy of the Clouds system is that the fundamental tools needed for the development of distributed applications are (1) a mechanism for distributed data access and (2) support for dealing with component failures. The object mechanism described above is designed expressly to support location-transparent data sharing. Processes interact not by passing messages to one another, but rather by accessing a shared object. This approach allows the processes to directly share the common part of their collective state rather than to attempt to communicate state changes directly to one another via messages.

Since the state of a computation is captured by the set of objects it access and modifies, it is important that component failures do not lead to inconsistent states in which some but not all of a list of related changes to objects have been completed. The atomic action mechanism provides such an assurance. An atomic action can be defined to consist of any number of operation invocations on a set of objects.

The project consisted of two research tasks. The goal of each task was the production of a technical guidebook outlining and analyzing tools and techniques for the development of fault tolerant software for distributed computing systems.

The title of Task 1 was "Programming Techniques for Resilience and Availability." The work of this task focused heavily on problems related to replication, since replication is the key ingredient of any scheme to provide highly available applications or services. Issues discussed in the guidebook include defining resilient data areas, naming replicas, locking in the presence of replicas, state propagation to replicas when actions commit, and fault tolerant action execution. Much of the discussion is in terms of the Aeolus language which we use to program objects for the Clouds system.

The title of Task 2 was "Action-Based Programming for Embedded Systems." The major issue addressed by this task is the seeming incompatibility of the idea of large-grained atomic actions with the irreversible operations frequently performed by embedded systems. Substantial consideration is given to the problem of preserving information about irreversible operations so that recovery mechanisms invoked by action aborts (or exceptions) can produce a meaningful system state, though not the same state as would be produced by a pure atomic action mechanism.

1.1. Applicability to Existing Systems

The resilience work in Task 1 focuses on features in the Aeolus language designed to support the definition of resilient objects. Its major point of general applicability is in how it relates to the more general concept of checkpointing. It illustrates the value and power of allowing a programmer to specify what must be checkpointed and how it is to be recovered.

The availability work in Task 1 is again somewhat specialized for the Clouds environment. However, it should be viewed as a model for the importance of allowing a mixture of contribution by the system and the programmer. The basic idea of the solution presented is that the system provides a basic framework and supporting mechanisms for availability while the programmer contributes policy implementations that are customized for a particular application.

The embedded system work of Task 2 has a more direct general application. It generalizes the atomicity concept by integrating forward and backward recovery, thus removing the incompatibility between the (generalized) atomicity concept and irreversible operations.

A common thread through all of the results is the importance of providing ways for a programmer to use application semantics in developing customized recovery, resilience and availability solutions, while at the same time providing the most powerful supporting mechanisms possible.

2. Programming Techniques for Resilience and Availability

In keeping with the title of this task, the most significant results presented in the Task 1 guidebook concern language features for resilient types and availability specifications. In keeping with our concern for providing powerful supporting mechanisms, it is significant to note that both of these features are *declarative*. Our intent is to allow a programmer, as far as possible, to specify resilience and availability requirements, leaving detail work to a compiler and runtime library.

Resilience and availability are crucial to our basic goal: fault tolerant software for distributed computing systems. By *resilience* we mean the survivability and consistency of data despite crashes and other detectable faults. We define *availability* to mean accessibility of data despite network partitions or failures of some sites in a distributed system. Together with a mechanism that ensures *forward progress* (continued execution of jobs despite failures), these properties provide fault tolerance.

Resilient types are a mechanism for specifying customized update and recovery mechanisms in an object designed to be modified by atomic actions. Such modifications imply that multiple versions of the object must be maintained while uncompleted actions exist. Use of customized operations based on object semantics in this context allows far more efficient use of atomic actions than would be possible if a generalized recovery scheme were used. In the later case, a copy of the entire data space of the object would have to be made for each active version of the object. Use of the resilient type technique allows all versions to be represented within a single address space.

The features for availability support presented in the guidebook are collectively known as *distributed locking*. They deal with support for managing replicas of an object in order to increase the availability of the service provided by the object. Using distributed locking, a programmer first writes a definition and implementation of an object as if only a single instance of the object was going to exist. (Resilient types might be used in the object implementation.) The programmer then writes an availability specification for the object, specifying the number of replicas, the the replication control policies to be used, and the relative availabilities of the modes of each lock type specified in the object. The most significant aspect of this specification is the replication control policy part. It allows the programmer to designate how concurrency control and consistency maintenance are to be performed, considering, as usual, object semantics. The mechanisms designated may be either taken from system libraries or supplied by the programmer.

3. Action-Based Programming for Embedded Systems

The work performed for this task was based on the idea that embedded systems include *irreversible operations*, that is, operations that interact with the physical environment. The performance of such operations appears incompatible with the concept of an atomic action, since atomic actions rely on rollback (or more generally, backward recovery) to restore their initial state in the case of a failure. The work we have done generalizes the atomicity concept by integrating forward and backward recovery, thus removing the incompatibility between the (generalized) atomicity concept and irreversible operations.

The solutions developed involve software recovery techniques presented within the framework of action-based programming. The recovery techniques described in the handbook represent a synthesis of exception handling and action-based programming. Exception handlers are associated with individual units of work (actions) rather than with procedures or objects. The exception handlers have access to system services not otherwise accessible in a program. These services are used to achieve appropriate forward recovery. To emphasize the nature of these enhancements, exception handlers are termed

recovery handlers.

A more general unanticipated result of our work is that the approach we present can be used not only to increase the fault tolerance of a software system, but also to simplify management and maintenance of the system. For example, if actions are robust, it will be possible to bring an individual machine down for maintenance without extensive coordination. A robust action will abort when the site goes down and either restart when it comes up or make alternative arrangement in the interim. Our approach can also be used to support software maintenance and upgrades. We describe how recovery handlers can remap the code and data windows of the associated action during recovery. This mechanism provides on-line access to backup versions of software and can be used to transfer control from the old version to a new version of the code for an action.

4. Appendices

The following papers based on and related to work performed during the course of this project are included as appendices:

"Fault Tolerant Computing in Object Based Distributed Systems" by Mustaque Ahamad, Partha Dasgupta, Richard J. LeBlanc, and C. Thomas Wilkes. From the Proceedings of the Sixth Symposium on Reliability in Distributed Software and Database Systems (March, 1987).

"Distributed Locking: A Mechanism for Constructing Highly Available Objects" by C. Thomas Wilkes and Richard J. LeBlanc. An abbreviated version of this paper will appear in the Proceedings of the Seventh Symposium on Reliability in Distributed Systems (October, 1988).

"The Clouds Distributed Operating System" by Partha Dasgupta, Richard J. LeBlanc and William F. Appelbe. From the Proceedings of the 8th International Conference on Distributed Computing Systems.

Appendix A

Fault Tolerant Computing in Object Based Distributed Operating Systems

Mustaque Ahamad, Partha Dasgupta, Richard J. LeBlanc, & C. Thomas Wilkes †

School of Information and Computer Science
Georgia Institute of Technology, Atlanta, GA 30332-0280

Abstract

Replication of data has been used for enhancing its availability in the presence of failures in distributed systems. Data can be replicated with greater ease than generalized objects. We review some of the techniques used to replicate objects for resilience in distributed operating systems.

We discuss the problems associated with the replication of objects and present a scheme of replicated actions and replicated objects, using a paradigm we call PETs (parallel execution threads). The PET scheme not only exploits the high availability of replicated objects but also tolerates site failures that happen while an action is executing. We show how this scheme can be implemented in a distributed object based system, and use the *Clouds* operating system as an example testbed.

1. Introduction

A distributed system consists of many computers which are connected via communication links. The increased number of components (i.e., machines, devices and communication links) increases the chances of a failure in the system (or decreases the mean time between failures). Guarding against the effects of failures is one of the key issues in distributed computing. In this paper, we discuss

† This research was partially supported by NASA under contract number NAG-1-430 and by NSF under contract number DCS-84-05020.

Authors' Address:
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332
Phone:
(404) 894 2572
Electronic Address:
{mustaq,partha,rich,wilkes} @ Gatech.edu
{akgua,allegra,hplabs,ihnp4}!gatech!{mustaq,partha,rich,wilkes}

approaches that provide forward progress despite the failure of some components in a distributed computing system.

Our model of the distributed system is a prototype under development at Georgia Tech named *Clouds*. *Clouds* is a decentralized operating system providing location transparency, transactions, and robustness in an object based environment. In this paper, we present a review of known techniques for fault tolerance using replication. Then we discuss the salient features and architecture of *Clouds*. Finally, we present mechanisms needed for replication, probes, and parallel action threads for providing fault tolerant computing in *Clouds*. We discuss the pitfalls and the solutions to the problem of providing replication of objects having a general structure, which is more complex to achieve than replication of *flat* data (data that is accessed through read and write operations, such as files).

2. Replication Techniques for Database Systems

The use of replication to enhance availability was first studied in the area of distributed database systems, and was later adopted in the area of distributed operating systems.

2.1 Concurrency Control of Replicated Data

One of the main issues in handling of replicated data in database systems is to maintain consistency. This is achieved by concurrency control protocols. The concurrency control and recovery techniques for replicated data are summarized by Wright.[Wrig84] He classifies these methods as *conservative (pessimistic, blocking)* and *optimistic (non-blocking)*.

Conservative Concurrency Control Methods Examples of conservative methods are voting schemes,[Giff79, Thom79] primary copy methods,[Ston79] and token-passing schemes.[LeLa78] These methods ensure consistency of the replicated data by requiring access to a special copy or a set of copies of the data. Primary copy methods allow access to a copy during a network partition only if the partition possesses the designated primary copy of the data. Token-passing schemes are an extension of primary copy methods. A token is passed among sites holding a copy of data, and the copy at the site currently holding the token is considered the primary copy. In the voting schemes, each copy of the data is assigned a (possibly different) number of votes and a partition possessing a majority of the votes for that object may access it. The conservative schemes are called *blocking* since the data is not available at a site in a partition which does not possess the primary copy (or token or majority of votes). Thus, the access must block until the partition is ended, even if a copy of the data is available in the partition. Indeed, under these schemes it is possible that *no* partition may have access to the data.

Optimistic Concurrency Control Methods The optimistic methods do not seek to ensure global consistency of replicated data during partitions.[Davi81, Davi82] Thus, accesses are not blocked if a replica of the data is available in the partition in question. Rather, inconsistencies in the replicas are resolved by use of backouts or compensatory actions during a merge process, once the partition is ended. It is assumed that the number of such inconsistencies will be small (hence, *optimistic*). However, tradeoffs may be made between consistency and availability. For example, the *Data-Patch* tool for designing replicated databases[Blau82, Garc83] assumes that, rather than strict consistency, a reasonable view of the database should be maintained to enhance availability.

3. Replication in Operating Systems

Research in database systems has been limited to consideration of flat data, and as we show later, the generalization to replication of objects having arbitrary structure leads to many problems. These include the mechanisms used for the copying of state among replicas and having to deal with multiple instances of a single operation invocation (or a procedure call). The distributed operating systems that provide replication of objects or abstract data types include the Eden system developed at the University of Washington, the ISIS system at Cornell, and the Circus replicated call facility built on top of Unix. The replication of abstract data types has also been studied by Herlihy.

Eden The Eden system[Alme83] has been operational at the University of Washington since April 1983. Support for replication in the Eden system has been studied at both the kernel level and the object level. The kernel level implementation of replication support is called the *Replect* approach (for replicated Ejects, or Eden objects), while the object level implementation is called *R2D2* (for Replicated Resource Distributed Database). Both implementations use quorum consensus for concurrency control.

ISIS The ISIS system developed at Cornell[Birm84, Birm85] supports *k-resilient* objects (operations on such an object survives up to k site failures) by means of checkpoints. This system provides both availability and *forward progress;* that is, even after up to *k* site failures, enough information is available at the remaining sites possessing the object replicas that work started at the failed sites can continue at these remaining sites. This is accomplished through a *coordinator-cohort* scheme, where a transaction executes at the coordinator site and the updates it performs on any objects are propagated to the cohort replicas. one replica acts as master during a transaction to coordinate updates at the other slave replicas (cohorts). The choice of which replica acts as coordinator may differ from transaction to transaction. The object state is copied from the coordinator to the cohorts. We call this method of state propagation *cloning*. This operation has been described as propagating a checkpoint of the entire coordinator,[Birm84] or, in a more recent paper, as propagating the most recent

version in a version stack.[Birm85]

In ISIS, a transaction is not aborted when a machine on which its coordinator is running fails (transactions are usually aborted only when a deadlock situation arises). Rather, the transaction is resumed at a cohort from the latest checkpoint. This cohort becomes the new coordinator. Operations which the coordinator had executed after the latest checkpoint took place must be re-executed at the new coordinator.

Circus Cooper has investigated a *replicated procedure call* mechanism called Circus which was implemented in UNIX.[Coop85] In Cooper's scheme, although replicas of a module have no knowledge of each other, they are bound (via run-time support) into a server called a *troupe* which may be accessed by clients. (The client knows that the server is replicated.) A module in Circus may have arbitrary structure, containing references to other modules. However, the module is currently required to be deterministic. His scheme uses *idemexecution* (operation execution at each replica) for state propagation. When a troupe accesses an external troupe, results of operations on modules of the server troupe are retained by the callees. These results are associated with *call sequence numbers*, and are returned when subsequent calls by the replicas of the caller troupe with the same sequence numbers are encountered. This avoids the inconsistencies that can be caused by multiple executions of the same call.

Herlihy's Work Herlihy[Herl84] uses semantic knowledge of arbitrary abstract data types (objects) to enhance the quorum consensus concurrency control method. Analysis of the algebraic structure of data types is used in the choice of appropriate intersections of voting quorums.

4. Basics of the Clouds Operating System

Clouds is a distributed operating system that supports *objects* and *actions*. The rest of this paper deals with a set of techniques that implement generalized replicated objects in the framework of the *Clouds* operating system. We discuss the salient features of *Clouds* in this section. For a more detailed description, the reader is referred to [Dasg85].

Figure 1 shows the hardware configuration of the *Clouds* prototype. The *Clouds* operating system provides support for the following facilities:

Distribution *Clouds* has been designed with loosely coupled distribution in mind. The hardware architecture consists of a set of general purpose machines connected by an Ethernet. The software architecture is a set of cooperating sub-kernels, which implement a monolithic view of the distributed system.

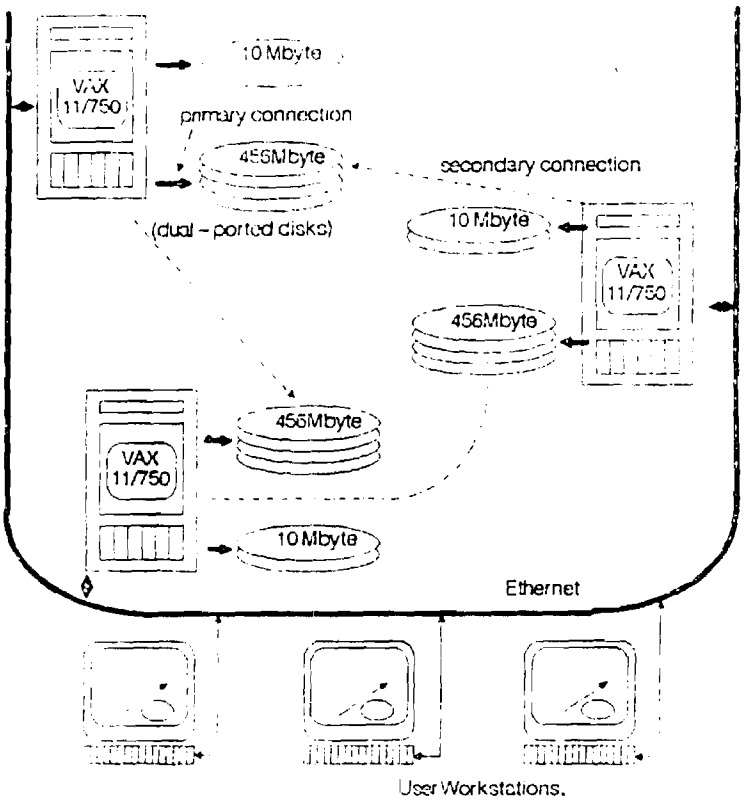

Figure 1. The Clouds Hardware Configuration

Object Based All system components, services, user data, and code are encapsulated in objects. The object structure is shown in Figure 2. The *Clouds* universe is a set of objects (and nothing but objects). An object is a permanent entity, occupying its own virtual address space. Processes can weave in and out of objects through entry points defined in the object space. The only way to access data in an object is to use a process that executes the code in the object via an entry point.

Location Independence The *Clouds* objects reside in a flat, system-wide name space (the system name space is flat, the user name space need not be). There are no machine boundaries. Any process that has access to an object can invoke an operation defined by the object. This creates a unified view of the system as one large computing environment consisting of objects, even though each site in the system maintains a high degree of autonomy.

Synchronization Objects are sharable, that is several processes can invoke the object concurrently. This can pose synchronization problems. *Clouds* implements an automatic as well as custom synchronization support for concurrent access to objects. (Automatic synchronization uses two-phase locking, using read and write locks. Custom synchronization is the responsibility of the object programmer.)

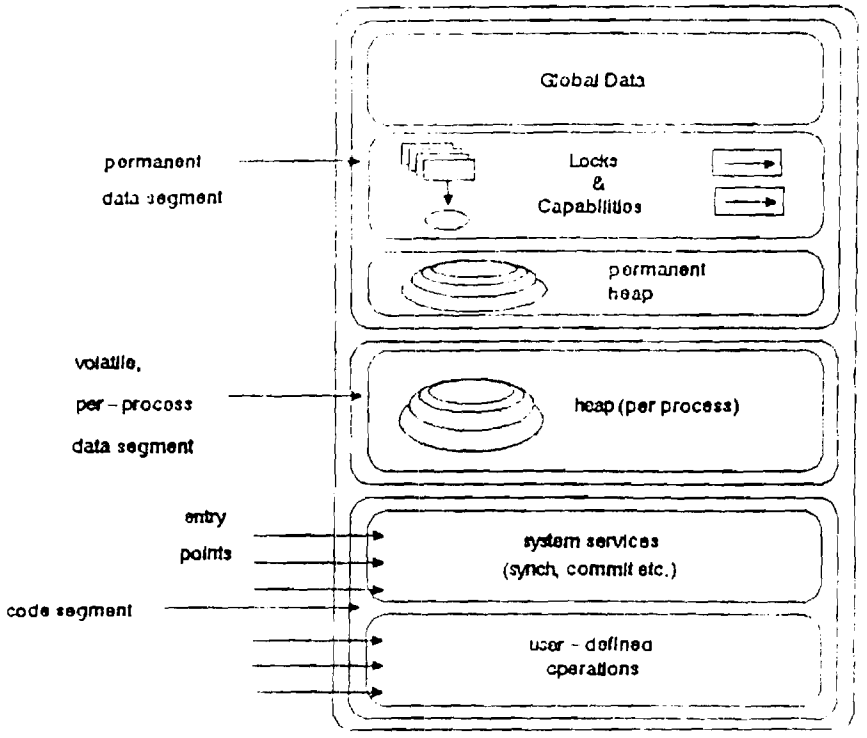

Figure 2. The Clouds Object Structure

Actions To prevent inconsistency in the data stored in objects, *Clouds* supports top-level and nested actions. Two-phase commit it used to ensure that all objects touched by an action are either updated successfully on a commit or are rolled back in case of explicit aborts or failures. The action management system tracks the progress of actions and maintains information about objects touched by the action and its subactions. The action management system uses the mechanisms provided by the recovery management component of the *Clouds* kernel, for performing the commit or abort operations when a action terminates or fails. Recovery management is implemented as part of the storage manager.

Clouds is designed to support a high degree of *fault-tolerance*. The mechanisms that provide this support are the topic of discussion in the rest of this paper. The following section discusses the approaches.

5. Fault Tolerance

One of the basic goals that motivated the design of *Clouds* was achieving fault tolerance. Several of the mechanisms currently supported by *Clouds* are geared to this end. Thus, we believe it is an ideal environment for building a fault tolerant system. We review some of the low level details that provide such support.

1. The object invocation strategy was designed for fault tolerant systems. When a process invokes an object (using its capability), and the object is not available locally, a global search-and-invoke is initiated.[Spaf86] This will successfully invoke the object if it is *reachable*. Failure of any site not

containing the object will not affect the invocation. The invocation will also find the object, if reachable, irrespective of where it is located, even if it was moved around in the recent past. Migration, failure, creation and deletion of objects etc. do not adversely affect the invocation mechanism.

2. All disk systems are dual-ported (or if possible, multi-ported). If a site fails, the disks belonging to the failed site are re-assigned to other working sites. Due to the location search-and-invoke mechanism, this switch can be done on the fly, and the objects that were made inaccessible due to the failure become accessible.

3. Users are not hard-wired to the sites, but are attached to logical sites through a front-end Ethernet (multiple Ethernets may be used for higher reliability, without changing our algorithms or architecture). If the site the user is attached to fails, some other site takes over and the user still has access to the system.

4. The system maintains consistency of all data (objects) in the system by using the atomic properties of actions (or transactions). *Clouds* implements nested atomic actions. This is the function of the action management system, which uses the synchronization and recovery provided at the kernel level. The commit and abort primitives are implemented in the kernel,[Pitt86] and the action manager implements the policies. Nested actions have semantics similar to that defined in[Moss81] and are used to firewall failed subactions.

All these mechanisms provide a certain degree of fault tolerance, that is, the system is not affected adversely by failures. Some actions are aborted, but the system as a whole continues functioning in spite of site failures. Though dual porting of disks does simulate some replication (that is, if a site fails, the data stored at the site is still available through an alternate path), this mechanism is not completely general because it can not tolerate media crashes. Also, actions executing on the failed site are forced to abort.

The action management scheme provides backward recovery and ensures that all data in the system remain consistent in spite of failures. However, this does not guarantee forward progress, as failures cause actions to abort. Fault tolerance should imply some guarantee of forward progress, that is an action should be able to continue in spite of a certain number of failures. We now discuss strategies that guarantee forward progress despite failures.

5.1 Primary/Backup Actions and Probes

One of the methods that allows fault tolerant behavior is the use of the primary/backup paradigm for actions. This paradigm is also used for fault-tolerant scheduler, monitor, and other subsystems requiring some degree of reliability.[McKe84,Dasg86] In this scheme, a fault-tolerant action is really two actions,

one being the primary, which does the work, and the other being a backup, which is a hot standby. The primary and backup use probes to ensure both are up. If the primary fails, the backup takes over (and creates a new backup). If the backup fails, the primary creates a new backup.

The primary/backup system can be implemented using the *Clouds* probe management system. In *Clouds*, a probe can be sent from a process to another process or an object. The probe causes a quick return of status information of the recipient. Probes work synchronously, and use high priority messages and non-blocking routines so that the response time is practically guaranteed. This allows use of timeouts to check for reachability or liveness.

If a particular object is unavailable due to some failed component (even though we have dual ported disks), both the primary and the backup actions are doomed to fail. Thus the primary/backup scheme has to be augmented with increased availability of objects. Replication is the well known technique for achieving higher availability of data.

5.2 Replication of Objects

Maintaining consistency of replicated data (i.e., files) is simpler than maintaining consistency of replicated objects because only the read and write operations are provided to access data. Objects, on the other hand, are accessed through operations defined in the objects, which in turn can call operations defined in other objects. This gives rise to the following problems:

1. Due to non-determinism, the same operation invoked on two identical copies of an object may produce different results. Thus non-determinism cannot be handled in the Circus system, because it uses idemexecution.

2. Due to the nested nature of the objects, two copies of a replicated object may make a call to a non-replicated object, causing two calls where there should have been one. This can happen in the ISIS system when the coordinator crashes and some other site becomes the coordinator. In Circus this happens when the caller object is replicated.

3. Maintaining varying degrees of replication of objects produces a fan-in fan-out problem that is not easy to handle. Also, the naming scheme for replicated objects presents a non-trivial problem.

The generality of the abstract object structure supported by *Clouds* poses problems for replication methods which are not presented by objects of lesser generality. The problem lies in the possibility of the arbitrarily complex *logical* nesting of *Clouds* objects. Although *Clouds* objects may not be *physically* nested (that is, one object may not physically contain another object), an object may contain a capability to another object. If object *A* creates another object *B*, and retains sole access to *B*'s capability (by refraining from passing the capability to other objects

and also not registering the capability with the object filing system [OFS]), we say that object *B* is *internal to* object *A*. The internal object *B* may be regarded as being *logically* nested in object *A*. If, on the other hand, object *A* passes *B*'s capability to some object not internal to *A*, or if *A* registers *B*'s capability with the OFS, we say that *B* is *external* to *A*. An external object is potentially accessible to objects that may not be internal to the object's creator.

Problems arise with replication schemes when internal and external objects are mixed together in the same structure, i.e., when an object may contain capabilities to both internal and external objects. These problems are associated with the method which is used to propagate the state of a replicated object among its replicas. External objects cause problems when idemexecution is used to propagate state changes among replicas. If the replicated object invokes an operation on an external object (e.g., a print queue server), then under idemexecution, that operation will be executed by each replica. If the operation being performed on the external object is not idempotent, this can cause serious problems (e.g., multiple submissions of a job to the print queue). Also, trouble may arise when idemexecution is used if the operation on the external object is non-deterministic (for instance, random number generation, or disk block allocation among multiple concurrent processes).

On the other hand, internal objects cause problems when cloning is used to propagate state. For example, assume that each replica of an object creates a set of internal objects. Then, when an operation is performed on one of the replicas, its state under cloning is copied to each of the other replicas. However, since the capabilities to the internal objects of the replicas are contained in their states, each replica now contains capabilities to the internal objects of the replica at which the operation was actually executed. Thus, the information about the internal objects of the other replicas is lost.

6. Replication Mechanisms

6.1 Replicated Actions

We have developed a scheme called *replicated actions*. Each replicated action runs as a nested action and has its own thread of execution. Each thread of control is called a *Parallel Execution Thread* or PET. The degree of the replicated action is the number of PETs that comprise the action. The degree is determined statically at the the time the top level action is created. If all objects touched by the action are replicated *k* times and the degree of the replicated action is also *k*, we can have each PET executing on a different copy of the object.

Briefly, the PET scheme sets up several parallel, independent actions, performing the same task, using a possibly different set of replicas of the objects in question. These actions follow different execution paths, on different sites, but only one of them is allowed to commit. The scheme is depicted in Figure 3, and its

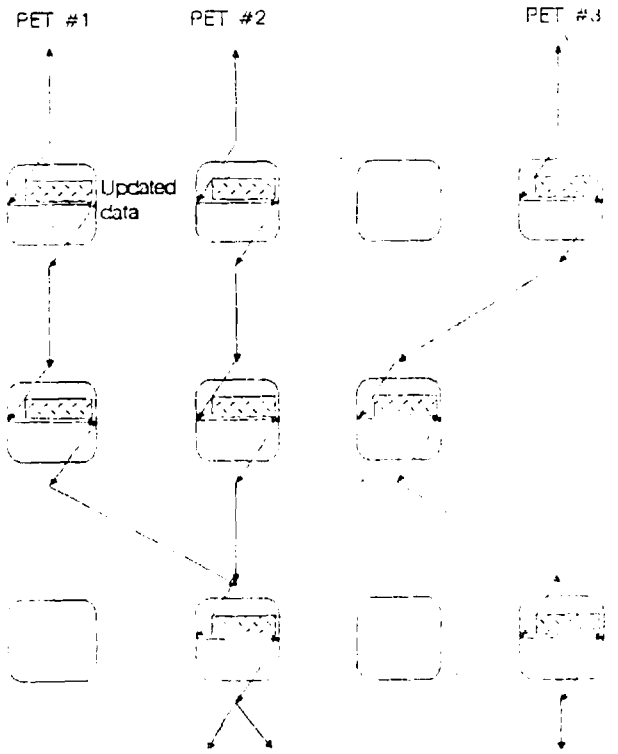

Figure 3. Parallel Execution Threads of 3-degree

implementation details are presented in Section 6.4.

The PET scheme for replicated objects has several advantages. Firstly, up to k-1 transient failures (in a PET scheme with k threads), are automatically handled because the remaining PETs will commit the action. This contrasts with the ISIS scheme in which one of the sites having a replica has to detect the failure of the coordinator and assume responsibility for the execution of the action. However it is possible for an action in ISIS to commit while all the PETs may abort in our scheme. The possibility of this happening is considerably reduced as the degree of the PETs are increased. Thus this scheme presents a trade off between computation and replication (overhead) and the degree of fault tolerance.

A replica of an object that is replicated k times can receive multiple calls (as in ISIS and Circus) when the PET degree is more than k. Thus a replica has to retain results to avoid executing the same call operation again. However a caller will not receive multiple results as in Circus and we do not have to collate the returned results. Also since only a single PET is allowed to commit, cloning is used for state copying and non-deterministic operations do not cause inconsistent state in the replicas. The problem of internal (or nested) objects is solved by a modification of the capability (naming) scheme, which is described below.

6.2 Naming Replicated Objects

Replicated objects and actions provide support for guaranteeing forward progress when system components fail. This introduces the problem of naming replicated

objects. In *Clouds*, the system uses a capability based naming scheme. A capability is a system name which uniquely identifies one object in the distributed system. Under this scheme, a *k*-replicated object is named by *k* different capabilities. This makes naming considerably more difficult, and since capabilities are stored within an object, state copying via cloning causes the problems described earlier.

To solve this we propose a minor modification to the capability scheme. When replication is supported by the kernel, at the user level, all copies of the replicated object have the same capability, and thus one capability refers to a set of objects. A flag in the capability tells the kernel that the capability points to a set of replicas of the object.

The kernel can then append a *copy number* to generate unique references to the objects. The kernel uses the <capability:copy-number> pair to invoke operations. Thus the kernel can choose to invoke the appropriate copy (or several copies) depending upon the replication algorithms used to resolve an invocation on a replicated object.

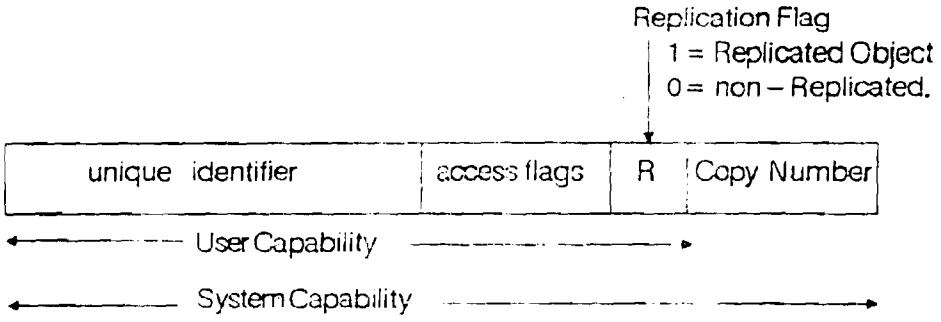

Figure 4. Capability Scheme for Replicated Objects

Since all references to the object, as far as the program is concerned, are still made through a unique capability, which points to all the copies, any naming problems at the user level disappear (when replication is supported by the kernel). Constructing the <capability:copy-number> pair can be effectively handled at the kernel level, using one of several techniques. (For example, the copy number 1 is always valid, and this copy, as well as other copies, contain information about the total number of copies, and thus all copies are accessed by the range 1..max.) This scheme is depicted in Figure 4.

6.3 Invocation of Replicated Objects

The invocation scheme for replicated objects has to follow the scheme outlined above. The kernel interface handles invocation as follows. For simplicity, in this section we will assume all the actions have only one thread of control (1-PET). We will generalize the scheme in the next section.

A process executing on behalf of an action requests the invocation of an operation defined by an object. The kernel examines the capability and detects whether the object is replicated or not. If it is not replicated, the invocation proceeds as a normal *Clouds* invocation. If the capability points to a replicated object, the kernel has to choose one of the replicas. If a local copy of the object is available, the kernel invokes the local copy, else it tries to invoke any one copy, by appending the copy number and sending out an invocation request on the broadcast medium. Typically, the kernel chooses copy number 1, and if that fails it tries subsequent copies. This sequential searching is not necessary, as the kernel can use previous history to decide which replica to use.

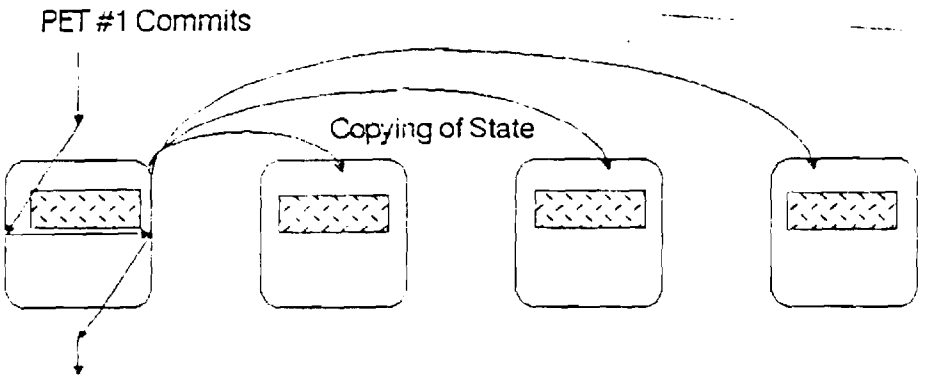

Figure 5. State Copying on PET Commit

Once a replica is used for an action, the kernel takes note of that, and stores it with the action id, and all later invocations are directed to that replica. Thus only a single replica of each replicated object is used to execute one action. The other replicas are not touched, until the action decides to commit. When an action commits, the replica it touched is copied to all other replicas. This is done by copy requests from the action management systems to all the replicas (using the copy number scheme). All accessible replicas are updated and their version numbers updated. (Note that if the source object has a copy number lower than a replica, the action has to be aborted.) The version copying strategy is shown in Figure 5.

The version numbers are also used to bring failed sites up-to-date on startup. On startup, all replicas at the site having version numbers less than the highest version number on the network are reinstated.

6.4 Handling PETs

The above scheme using 1-PET execution is prone to failures in certain cases. These include cases where a replica becomes unavailable after it has been invoked, the replica invoked was not up-to-date and when the site coordinating the action fails.

The N-PET (N>1) case decreases the chances of transaction abort due to the transient failures described in the earlier paragraph. All the separate PETs have

different co-ordinating sites and execute independently.

When the first thread invokes a replicated object, the invocation proceeds as above, that is a replica is chosen to service the action. The second thread also proceeds similarly, but a different replica is chosen. The replica choice does not have to be different, but the reliability increases if they are, so we use a random choice scheme. Note that the same object is chosen (as there is no choice) if the object is not replicated. Multiple invocations of the same object, due to multiple threads of control are handled by a collator. The commit phase is however different.

In this scheme, ONLY one PET can be allowed to commit. If more than one PET reaches commit point, each PET issues a pre-commit, which checks if all the primary copies it touched are still available. If any thing is not reachable, the PET aborts. Of the remaining PETs any one has to be chosen to commit (In fact if all of them are allowed to proceed, they will overwrite each others results and may cause deadlocks during commit time.) The co-ordinating site with the highest site number wins the match and commits the PET that was associated with the site. The commit causes the replicas touched by this PET to be copied to all other replicas. The co-ordinating sites that lost the commit war, do not abort the PETs, but wait for the commit of the winner to be over. If the commit fails the co-ordinator with the next highest site number attempts the commit. (Note that the previous commit could have attempted to overwrite the replicas touched by this PET, but the pre-commit causes a special copy of all the replicas to be retained, and this copy is used for the commit.)

Transient failures cause failed PETs, but the chances of all PETs failing decreases as the number of PETs is increased. Also, failures during commit are taken care of, by the other PETs. Of course it is possible for all the PETs to abort, but the chances of this happening decrease as the replication degree and the PET degree is increased.

7. Concluding Remarks

There are two major contributions of this research.

1. The object replication scheme is not as straightforward as data replication. The capability scheme allows reference to a set of objects and the cloning technique ensures correct execution in spite of generalized and nested objects, as well as non-deterministic objects.

2. Replication enhances availability, that is, actions can be run on a system that has some sites or data missing due to failures. Handling transient failures are not possible in most replicated schemes, that is, if an action touches an object, and the object later becomes inaccessible, before the action commits, the action has to abort. Also, once an action has visited a site, the failure of that site before the action commits can lead to action failure. The PET scheme allows

the action to proceed, with high probability of success, in a unreliable environment, where sites fail and restart during the execution time of the action.

We are currently involved with designing the lower level algorithms and modifying the *Clouds* action management scheme to implement the PET method of providing fault tolerance in the *Clouds* operating system. This involves the implementation of the collators, the kernel primitives to choose the appropriate replicas, the mechanisms that ensure distinct PETS choose distinct replicas and so on. Once the implementation is complete, we will be able to experimentally study the reliability of this approach.

REFERENCES

[Alme83] Almes, G. T., A. P. Black, E. D. Lazowska, and J. D. Noe. "The Eden System: A Technical Review." TECHNICAL REPORT 83-10-05, University of Washington Department of Computer Science, October 1983.

[Birm84] Birman, K. P., T. A. Joseph, T. Raeuchle, and A. El-Abbadi. "Implementing Fault-Tolerant Distributed Objects." *PROCEEDINGS OF THE FOURTH SYMPOSIUM ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS*, Silver Spring, MD (October 1984): 124-133.

[Birm85] Birman, K. P. "Replication and Fault-Tolerance in the ISIS System." *PROCEEDINGS OF THE TENTH SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES* (ACM SIGOPS), Orcas Island, Washington (December 1985). (Also released as technical report TR 85-668.)

[Blau82] Blaustein, B., R. M. Chilenskas, H. Garcia-Molina, D. R. Ries, and T. Allen. "Partition Recovery Using Semantic Knowledge." (TECHNICAL REPORT), Computer Corporation of America, Cambridge, MA, November 1982.

[Coop85] Cooper, E. "Replicated Distributed Programs." *PROCEEDINGS OF THE TENTH SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES* (ACM SIGOPS), Orcas Island, WA (December 1985): 63-78. (Available as *Operating Systems Review* 19, no. 5.)

[Dasg85] Dasgupta, P., R. LeBlanc, and E. Spafford. "The Clouds Project: Design and Implementation of a Fault-Tolerant Distributed Operating System." TECHNICAL REPORT GIT-ICS-85/29, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985.

[Dasg86] Dasgupta, P. "A Probe-Based Monitoring Scheme for an Object-Oriented Distributed Operating System." *Proceedings Of The Conference On Object Oriented Programming Systems, Languages And Applications* (ACM SIGPLAN), Portland, OR (Sept. 1986): 57-66. (Also available as Technical Report GIT-ICS-86/05.)

[Davi81] Davidson, S., and H. Garcia-Molina. "Protocols for Partitioned Distributed Database Systems." *Proceedings Of The Symposium On Reliability In Distributed Software And Database Systems*, Pittsburgh, PA (July 1981).

[Davi82] Davidson, S. "An Optimistic Protocol for Partitioned Distributed Database Systems." Ph.D. Diss., Department of Electrical Engineering and Computer Science, Princeton University, 1982.

[Garc83] Garcia-Molina, H., T. Allen, B. Blaustein, R. M. Chilenskas, and D. R. Ries. "Data-Patch: Integrating Inconsistent Copies of a Database after a Partition." *Proceedings Of The Third Symposium On Reliability In Distributed Software And Database Systems*, Clearwater Beach, FL (October 1983).

[Giff79] Gifford, D. K. "Weighted Voting for Replicated Data." *Proceedings Of The Seventh Symposium On Operating Systems Principles* (ACM SIGOPS), Pacific Grove, CA (December 1979).

[Herl84] Herlihy, M. "Replication Methods for Abstract Data Types." Ph.D. Diss., Laboratory for Computer Science, Massachussetts Institute of Technology, Cambridge, MA, May 1984. (Also released as Technical Report MIT/LCS/TR-319.)

[LeLa78] LeLann, G. "Algorithms for Distributed Data-Sharing Systems Which Use Tickets." *Proceedings Of The Third Berkeley Workshop On Distributed Data Management And Computer Networks*, Berkeley, CA (August 1978).

[McKe84] McKendry, M. S. "Fault-Tolerant Scheduling Mechanisms." (Unpublished Technical Report), School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, May 1984. (Draft only.)

[Moss81] Moss, J. "Nested Transactions: An Approach to Reliable Distributed Computing." Technical Report Mit/Lcs/Tr-260, MIT Laboratory for Computer Science, 1981.

[Pitt86] Pitts, D. V. "Storage Management for a Reliable Decentralized Operating System." Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also

released as Technical Report GIT-ICS-86/21.)

[Spaf86] Spafford, E. H. "Kernel Structures for a Distributed Operating System." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as technical report GIT-ICS-86/16.)

[Ston79] Stonebreaker, M. "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES." *TRANSACTIONS ON SOFTWARE ENGINEERING* (IEEE) **5**, no. 3 (May 1979).

[Thom79] Thomas, R. H. "A Majority Consensus Approach to Concurrency Control for Multiple-Copy Databases." *TRANSACTIONS ON DATABASE SYSTEMS* (ACM) **4**, no. 2 (June 1979).

[Wrig84] Wright, D. D. "Managing Distributed Databases in Partitioned Networks." PH.D. DISS., Department of Computer Science, Cornell University, Ithaca, NY, January 1984. (Also available as Cornell University Technical Report 83-572.)

Distributed Locking: A Mechanism for Constructing Highly Available Objects

C. Thomas Wilkes*
University of Lowell

Richard J. LeBlanc, Jr.†
Georgia Institute of Technology

April 4, 1988

Abstract

Distributed Locking refers to a methodology for constructing replicated objects from single-site implementations in an action-based object-oriented system such as the *Clouds* project. It also refers to the mechanism provided to support this methodology in Clouds. This mechanism assumes no particular ***policy*** for control of replica concurrency and consistency; rather, it provides primitives with which a wide range of policies may be supported. Also, by use of extensions to the *Aeolus* systems programming language supporting ***replication events***, the specification of the availability properties of an object is abstracted from the object implementation. Thus, a replicated object may be constructed from a single-site implementation, or changes made in the policies used for control of a replicated object, with little or no change to the object implementation. Examples of the specification and use of the quorum consensus replication control policy using the Distributed Locking primitives are described.

---

*This work was performed while the author was with the School of ICS at Georgia Tech. Author's present address: Department of Computer Science, University of Lowell, One University Avenue, Lowell, MA 01854. Internet: wilkes@hawk.ulowell.edu

†Author's address: School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332-0280. Internet: rich%lynx@gatech.edu

Distributed Locking:
A Mechanism for Constructing Highly Available Objects

1. Introduction

Among the benefits claimed for distributed computing are improvements in system fault tolerance and reliability, and increased availability of data and services. The *Clouds* project at Georgia Tech is one of a number of recent proposals in which reliability in a distributed system is based on the use of *atomic actions*, a generalization of the transaction concept of distributed databases. As part of the Clouds project, we have designed and implemented a high-level language providing access to the synchronization and recovery features of the Clouds system; this language is being used to implement those levels of the Clouds system above the kernel level. It also provides a framework within which to study programming methodologies suitable for systems based on the action concept, such as Clouds. Among the properties needed by systems data structures, the design of which must be addressed by such methodologies, are *resilience*—survivability and consistency of the data despite crashes and other faults; and *availability*—increased possibility of access to data despite network partitions or failures of some sites in a multicomputer system. Together with a mechanism that ensures *forward progress*—continued execution of jobs despite failures, these properties provide *fault tolerance* in the system.

In this paper, we describe some of the results of a study of methods of achieving fault tolerance in the Clouds system, in particular achieving increased availability of objects in Clouds. The remainder of this introduction presents the problems explored by this work. Section 2 describes the model of distributed computation in which the problems posed by the research were examined (the Clouds system) and the tools which were used to address these problems (the Aeolus[1] programming language). In Section 3, we present a methodology for achieving available services by conversion of resilient single-site implementations into replicated implementations. A mechanism with which we proposes to support this methodology, called *Distributed Locking*, is also described in Section 3. In Section 4, we describe a linguistic feature for the specification of the availability properties of an object replicated via Distributed Locking. The language runtime support features (primitives) required to support Distributed Locking, as well as operating system support needed to support these features, are presented in Section 5. In Section 6, previous work in database systems is presented as well as work in the operating system area that is relevant to the author's research. Finally, the conclusions which we have drawn from this research are summarized in Section 7, as are plans for future extensions of this work.

The work described in this paper is, in general, concerned with situations in which sites fail by halting, that is, *fail-stop* failures [Schl83]; in particular, malicious activity by failed sites (so-called *Byzantine* failures) are not considered here.

1.1 The Need for Availability

Even if a computation is distributed, it is subject to a single point of failure if any of the data objects involved in that computation exist at only a single node. The provision of resilience alone cannot eliminate the problems caused by site or network failures; although inconsistencies introduced by such failures have been abolished, any objects existing only at a failed site are unavailable for the duration of the failure, and thus no computation may proceed which requires

1. Aeolus was the king of the winds in Greek mythology.

those objects. A method for eliminating these bottlenecks is *data replication*, that is, the maintenance of copies of an object at multiple sites.

The use of replication introduces the problem of maintaining the *consistency* of the individual replicas when operations are executed on them. A common requirement for consistency is that the replicated object maintain *single-copy semantics*, that is, that the state of each replica be consistent with that which would have been obtained had the object existed only at a single site and had the same sequence of operations been applied to it. This is achieved by a combination of a mechanism for controlling concurrency among the replicas, and of a mechanism for copying the state obtained by an operation execution among the replicas.

These mechanisms have been the subject of much study, both in the areas of database systems and of operating systems. Indeed, it has been found that single-copy semantics is too stringent a requirement in some applications. (See [Wilk87] for a discussion of previous work in this area.) However, most previous work on such mechanisms has been concerned with "flat" data, such as files. The unique problems posed for these mechanisms by the object construct used in systems such as Clouds are discussed in the following section; in so doing, we also introduce some terminology used in the remainder of this paper.

1.1.1 Problems of Replication in Object-Based Systems In the course of research on methods of achieving availability in object-based systems such as Clouds, we have found that the generality of the abstract object structure supported by Clouds poses problems for replication methods which are not presented by a less general, flat object structure (for instance, files or queues).

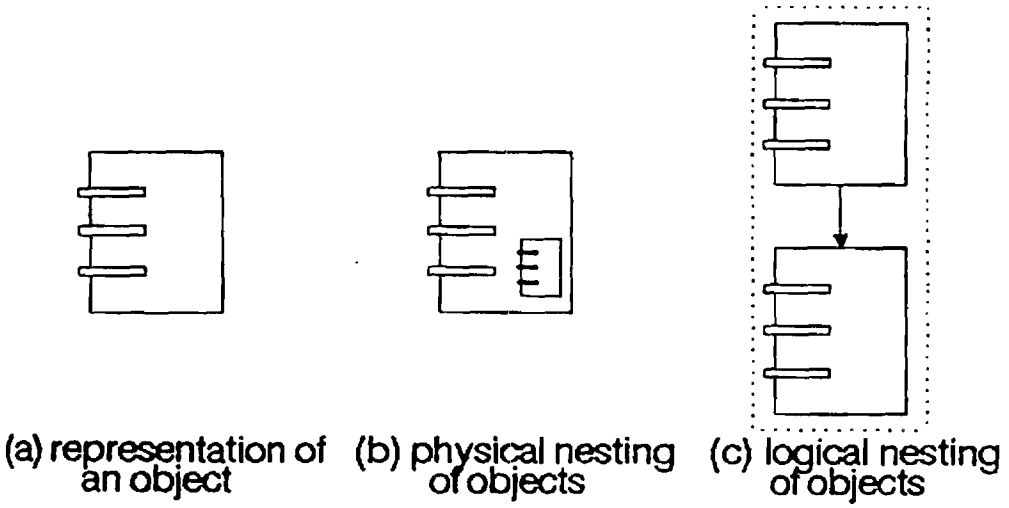

Figure 1. Pictorial Representation of Object Nesting

The problem lies in the possibility of the arbitrarily complex *logical* nesting of Clouds objects. Although Clouds objects may not be *physically* nested (that is, one object may not physically contain another object), an object may contain a capability to another object. If an object A creates another object B, and retains sole access to B's capability (by refraining from passing the capability to other objects, either explicitly or through an intermediary such as an object directory service), object B is said to be *internal to* object A. The internal object B may be regarded as being *logically* nested in object A. (A pictorial representation of physical and logical nesting is shown in Figure 1.) If, on the other hand, object A passes B's capability to some object not internal to A, or if A registers B's capability with an object directory service, B is said to be an *external* object; an external object is potentially accessible by objects not internal to the object

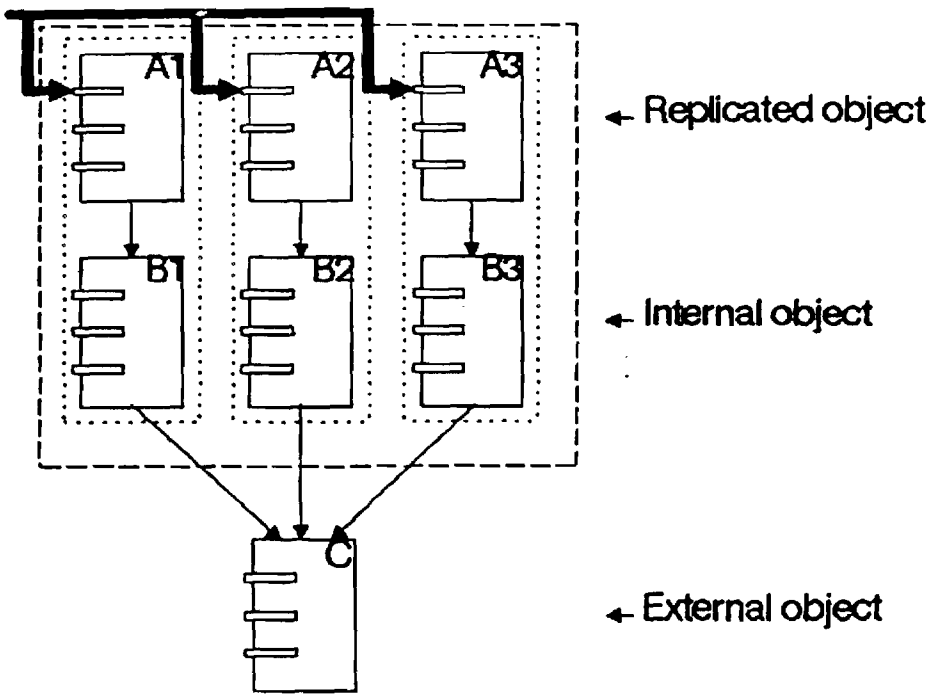

Figure 2. Replicated Object with Internal and External Object References

which created the external object.

Problems arise with replication schemes when internal and external objects are mixed together in the same structure, *i.e.*, when an object may contain capabilities to both internal and external objects. (An example of such an object is represented in Figure 2.) These problems are associated with the method which is used to propagate the state of a replicated object among its replicas. One such method is to execute at each replica the computation from which the desired state results; this scheme is called *idemexecution*. Another method is to execute the computation at one replica, and then copy the state of that replica to the other replicas; this scheme is called *cloning*. (Representations of the idemexecution and the cloning methods are shown in Figure 3.) Note that the scheme which is used to ensure that the replicas maintain consistent states (*e.g.*, quorum consensus) is not involved in these problems, and is considered separately in this investigation.

External objects cause problems when idemexecution is used to propagate state among replicas. If the replicated object performs some operation on an external object (*e.g.*, a print queue server), then—under idemexecution—that operation will be repeated by each replica. If the operation being performed on the external object is not idempotent, this can cause serious problems (*e.g.*, multiple submissions of a job to the print queue). Also, trouble may arise due to idemexecution if the operation on the external object is non-deterministic (for instance, random number generation, or disk block allocation among multiple concurrent processes).

On the other hand, internal objects cause problems when cloning is used to propagate state. For example, assume that each replica of an object creates a set of internal objects. Then, when an operation is performed on one of the replicas, its state—under cloning—is copied to each of the other replicas. However, the capabilities to the internal objects of the replicas are contained in their states; thus, each replica now contains capabilities to the internal objects of that replica on which the operation was actually performed, and the information about the internal objects of the

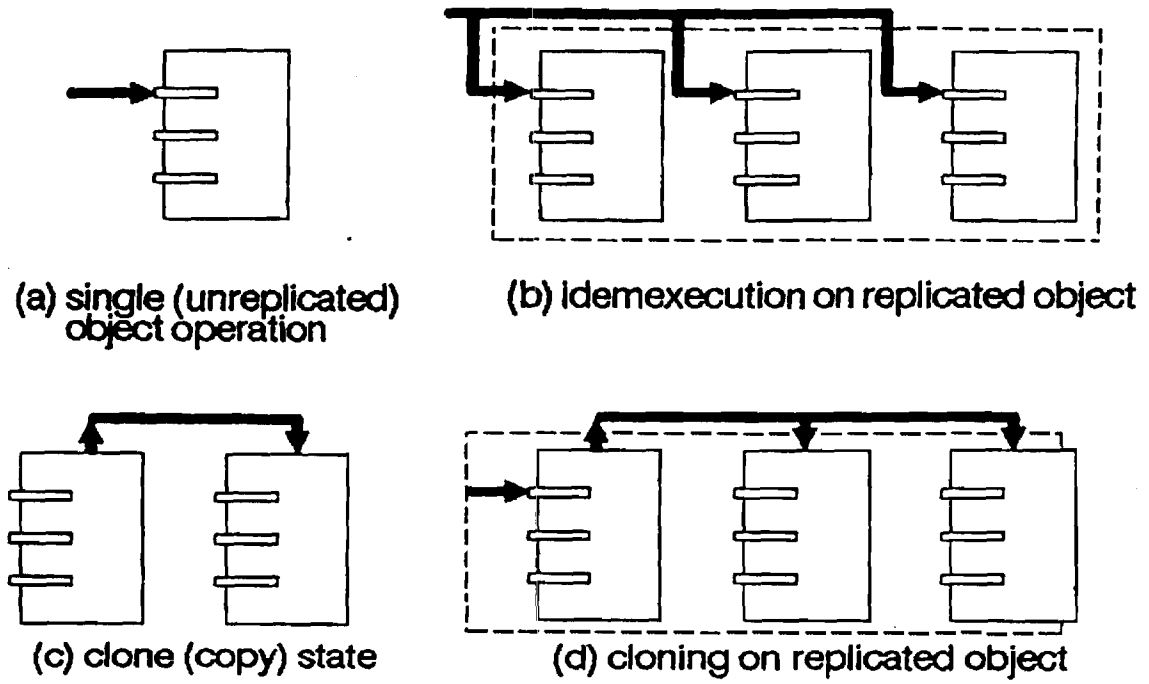

Figure 3. Replicated State-Copying Methods

other replicas is lost. This problem is illustrated in Figure 4. In Figure 4 (a), each replica has a capability to its individual internal object. In Figure 4 (b), an operation execution has taken place at the leftmost replica in the figure, and its state has been cloned to the other two replicas; the states of the other replicas now contain capabilities to the internal object of the leftmost replica rather than to their own internal objects.

1.2 The Need for Distributed Locking

In recent years, several researchers have presented algorithms that have explored the feasibility of trading consistency for availability in specific applications, or have taken advantage of semantic knowledge of typed objects to increase resilience or availability of these objects. (This related work is described in Section 6.) It has become clear from this research that, in certain applications, the ability to exploit trade-offs between consistency and availability, and to make use of the semantic knowledge of objects towards these goals, is not only feasible but desirable. It thus seemed inadvisable to limit the user to any pre-specified algorithm; none seemed sufficient to handle all of the potential applications. Accordingly, the focus of the research presented here changed to the question of how the various algorithms and techniques might be supported in the Clouds system in a general and efficient manner.

Features to support programming for resilience were introduced into the Aeolus testbed language at a relatively early stage, as these model closely the mechanisms provided by the Clouds kernel; these features are described in Section 2. Features to support programming for availability, on the other hand, were designed at a relatively late stage of this research. Our first attempts to program available objects in Aeolus soon convinced us, due to their *ad hoc* nature, of the desirability of linguistic support in this area. In these early attempts the manual addition of support for replication to an object originally designed as a single-site implementation was distressingly inelegant; an example of the result of this strategy is supplied elsewhere [Wilk87]. This experience suggested that a proper goal would be automation (to whatever extent possible)

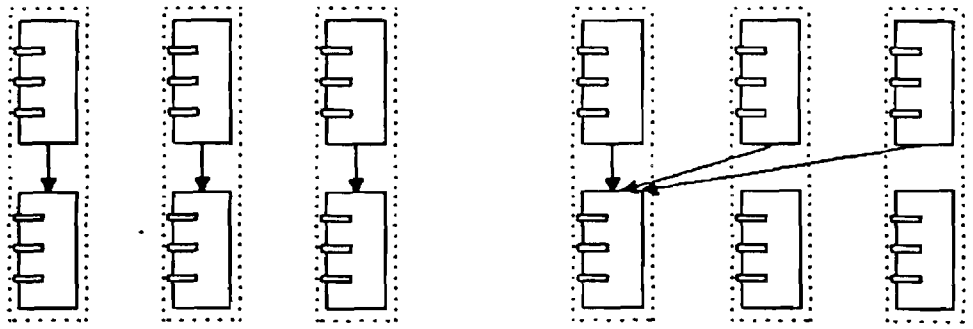

Figure 4. State Cloning with Internal Objects

of the process of deriving a replicated implementation of an object from its single-site implementation. The resulting *Distributed Locking* mechanism, described in Section 3, provides support for the control of concurrency and state-copying among replicas of an object while making no assumptions about the policies used for this control. The abundance of replication-control algorithms that has appeared in the literature in recent years, often taking advantage of the semantics of a particular application, makes it clear that limiting support to any particular policy would be undesirable. Rather, primitives are provided to support programming of custom replication-control policies which may take advantage of semantic knowledge of objects; however, options are provided for the automatic use of one of several common replication-control algorithms, if desired. This is in accord with the philosophy of the Clouds system as demonstrated by its treatment of the issues of synchronization and recovery. The linguistic support added to Aeolus to aid the programmer in the specification of the availability properties of an object is described in Section 4.

2. The Aeolus/Clouds Model

In this section, we provide an overview of the model of distributed computation embodied in the Aeolus/Clouds system. The background of the Clouds distributed operating system project, as well as the major concepts and facilities presented by the Clouds system, are presented here; a more complete description of the system may be found in a recent overview paper [Dasg87]. Also, the major features of the *Aeolus* language are described briefly.

2.1 The Clouds System

The Clouds distributed operating system project has been under development at Georgia Tech since late 1981; the central concepts were developed by Allchin and McKendry in a pair of early papers [Allc82, Allc83], and the Clouds architecture was described in full in Allchin's dissertation [Allc83a]. The goal of the Clouds project is the implementation of a fault-tolerant distributed operating system based on the notions of *objects*, *actions*, and *processes*, to provide an environment for the construction of reliable applications on unreliable hardware. The basic approach is to exploit the redundancy available in distributed systems which consist of multiple computers connected by high-speed local area networks. Such systems are called *multicomputers* or *computer clusters*. In Clouds, the notion of an *object* may be used to represent system components, such as directories or queues. A set of changes to objects may be grouped into an *action*, which corresponds roughly to the *transaction* concept of distributed database work, providing an "all or nothing" assurance of atomic execution (a property sometimes called *failure*

atomicity). The underlying support system ensures that, even if the actions extend across multiple machines, the changes will occur in totality or not at all. At this level, the support system, known as the *Clouds kernel*, is maintaining the *consistency* of the objects. It ensures that objects either reflect the effects of an action totally or not at all—no intermediate states are possible. This guarantee of an action's totality permits one to characterize the effects of hardware component failures: they cause actions to fail. Since a failed action is guaranteed to have had no effects on the objects with which it interacted, the action may be restarted without concern for potential inconsistencies it might have created.

Actions in Clouds go beyond the related notion of transactions in a database system. Rather than modelling all access to objects as simple reads or writes, the Clouds approach supports arbitrary operations on objects and allows a programmer to take advantage of operation semantics to increase concurrency, and thereby, performance. Through appropriate use of encapsulation, concurrent actions can be allowed to change objects without violating serializability.

A powerful feature of Clouds is the separation of the two components the traditional notion of the serializability of atomic actions, *failure atomicity* and *view atomicity*. Failure atomicity, as mentioned above, refers to the "all or nothing" property of atomic actions; view atomicity requires that the effects of an uncommitted action are not seen by other actions until commital occurs, thus avoiding the problem of "cascading aborts" of actions which have viewed intermediate states of an uncommitted action that later is aborted. This separation of the recovery and synchronization aspects of serializability allows the Clouds programmer to design objects that, while maintaining an appearance of serializability to the outside world, may violate strict serializability internally—in ways based on the programmer's knowledge of the object's semantics—in the interest of system efficiency.

Objects, actions, and processes are fundamental concepts supported by the Clouds architecture. To support these concepts, recovery and consistency are incorporated into the basic virtual memory mechanism [Pitt86, Pitt87]. Synchronization mechanisms to control the interactions of actions are also provided. It is with these capabilities that Clouds is meant to support the data integrity required for the implementation of reliable, distributed application programs.

The detailed design of the Clouds kernel is discussed in Spafford's dissertation [Spaf86]. A prototype of the Clouds kernel, also described by Spafford, has been implemented on a hardware testbed consisting of VAX® 750s connected by a 10Mbps Ethernet, several dual-ported disk drives, and Sun 3 Workstations® running UNIX®—also attached to the Ethernet—that provide a user interface to the Clouds system. The Clouds kernel is implemented "on the bare machine," that is, it is not implemented on top of some other operating system such as UNIX. Thus, the features of objects, actions, and processes have been implemented in the lowest levels of the kernel, allowing use of the Clouds concepts in the construction of the operating system itself. At these lowest levels, we attempt to avoid implementing *policies*, instead providing *mechanisms* with which policies may be constructed. Some policies are embedded in subcomponents of the kernel. The storage management system [Pitt86] implements support for action-based stable storage within the object virtual memory mechanism. The action manager [Kenl86] controls the interaction of actions with objects, including creation, committal, and abortion of actions, a

---

® VAX is a registered trademark of Digital Equipment Corp.

® Sun Workstation is a registered trademark of Sun Microsystems, Inc.

® UNIX is a registered trademark of AT&T.

time-based orphan detection facility. and support for lock-based synchronization. Those kernel subcomponents implementing policy are intended to be replacable with minimal changes to the rest of the kernel. For instance, the storage management system could be replaced with another implementing log-based recovery, or the action manager changed to support timestamp-based synchronization, without fundamental changes to other kernel subcomponents.

The Clouds system above the kernel level consists of a set of fault-tolerant *servers* which provide system services (such as object filing, job scheduling, printer spooling, and the like) to application programs. (It is for the construction of this level of the Clouds system that the Aeolus programming language was designed; the kernel itself has been implemented in the C language.)

The location-transparency and resilience mechanisms provided by the Clouds architecture are used to support the operating system itself and its services. Thus, the system itself is decentralized (in the sense that the system can survive the failure of any node) and resilient. The Clouds system may be considered to consist of a set of fault-tolerant objects which in combination provide a reliable environment for applications.

2.2 The Aeolus Programming Language

In this section we provide a brief overview of the Aeolus programming language. More complete discussions of Aeolus may be found in previous publications [Wilk85, Wilk86, Wilk87].

Aeolus developed from the need for an implementation language for those portions of the Clouds system above the kernel level. Aeolus has evolved with these purposes:

- to provide the power needed for systems programming without sacrificing readability or maintainability;
- to provide abstractions of the Clouds notions of objects, actions, and processes as features within the language;
- to provide access to the recoverability and synchronization features of the Clouds system; and
- to serve as a testbed for the study of programming methodologies for action-object systems such as Clouds [LeBl85].

The intended users of Aeolus are systems programmers working on servers for the Clouds system. Clouds provides powerful features for the efficient support of resilient objects where the semantics of the objects are taken into account; it is assumed that the intended users have the necessary skills to make use of these features. Thus, although access to the automatic recovery and synchronization features of Clouds is available, we have avoided providing very-high-level features for programming resilient objects in the language, with the intention of evolving designs for such features out of experience with programming in Aeolus.

2.2.1 Support For Synchronization Aeolus provides access to the action manager's support for synchronization via a *lock* construct. An unusual aspect of Aeolus/Clouds locks is that they are associated not with the specific data being locked, but rather with values in some *domain*. Thus, an lock is obtained for a value of an object, and not on the object itself. Thus, for instance, a lock may be obtained on a file name even if that file does not yet exist. Another interesting feature of Aeolus/Clouds locks is that they provide a mechanism for the specification of arbitrary locking *modes* and arbitrary compatibilities between the different modes, thus allowing the lock to be tailored to the specific synchronization semantics of a subset of object operations. For example:

```
type file_lock is lock ( read : [ read ], write : [] )
        domain is string( FILE_NAME_SIZE )
```

The declaration of `file_lock` defines a lock type over the domain of strings representing filenames, in which the usual multiple reader/single writer synchronization is specified by the compatibilities among the `read` and `write` modes of the lock.

All locks obtained during execution in the environment of a nested action are retained and propagated to the immediate ancestor of that action upon commitial unless explicitly released by the programmer. Locks obtained under an action are automatically released if the action aborts or successfully performs a toplevel commit. Thus, a two-phase locking protocol (2PL) is maintained, with violations to 2PL allowed (via explicit release of locks) if the programmer deems such violations acceptable. A lock is available to be granted under a nested action even if conflicting locks are held under one or more of the ancestors of that action, but not if conflicting locks are held under an action which is not an ancestor of the nested action [Allc83a]. The power of the Aeolus/Clouds lock construct in supporting user-defined synchronization lies in the specification of arbitrary locking modes, and arbitrary compatibilities between those modes, as well as the dissociation of locks from the locked variables.

2.2.2 Support for Objects The *object* construct provides support for *data abstraction* in Aeolus. A collection of related data items may be *encapsulated* within an object, which also may provide *operations* (procedures that operate) on the data. The only access to the data of an object is via these operations; thus, an object can strictly control manipulation of its encapsulated data, helping guarantee the invariants of the abstraction. The declaration of the object defines a type, called an *object type*, which may be used in the declaration of variables to hold capabilities to *instances* of that object type.

Aeolus provides a hierarchy of *object classifications* sharing a common implementation and invocation syntax which offers a trade-off of functionality and efficiency. The object classifications fall into two groups: the so-called Clouds object classifications (`autorecoverable`, `recoverable`, and `nonrecoverable`) may make use of the object management facilities and (for `autorecoverable` and `recoverable` types) the action management facilities, while the non-Clouds object classifications (`local` and `pseudo`) do not use any of the Clouds facilities for action or object management and provide data-abstraction facilities usable "locally" (without resorting to the system facilities supporting distribution of objects). On the other hand, the Clouds object classifications provide access to the support for data abstraction provided by the Clouds system when the expense of that support is warranted; the separate classifications of Clouds objects allow the programmer to specify the degree of support (and of incurred expense) required. The object classifications are described in more detail in the papers cited above; while the `autorecoverable` classification provides the paradigm most often presented by other action systems, that is, completely automatic recovery of the entire object state, the `recoverable` classification is of more interest here in that it allows the programmer to tailor object recovery based on the semantics of the object via mechanisms described below.

The global variables of an object are called collectively the object's *state*. In an object of class `recoverable`, part of the object state may be specified to be in a *recoverable area;* also, the programmer may specify an *action events part* and/or a *per-action variables part*. Recoverable areas, action events, and per-action variables are described below.

In order to allow the object to participate in its own creation and deletion, an object implementation part contains specifications of handlers for the so-called *object events*. The object events include the *init* or object initialization event, the handler for which is executed whenever an instance of the object is created by use of an allocator; the *reinit* or object reinitialization event, the handler for which is executed—if the object has registered its desire for

reinitialization with the action manager—when the system is reinitialized after a crash or network partition; and the *delete* or object deletion event, the handler for which is executed when the object instance is destroyed.

An invocation of an object operation looks much like a procedure invocation, except that, outside the implementation part of the object itself, an operation name must be qualified by the name of a variable representing an instance of that object type (or, for pseudo-objects, by the name of the object type itself). Thus, for an instance of a bounded-stack type, the programmer might write

```
stack_instance @ push( elem )
```

When an object invokes one of its own operations, however, the usual procedure call syntax is used.

Invocations of pseudo-object and local object operations have semantics essentially similar to those of calls to procedures local to a compiland. The situation is different for operations declared in objects which use the Clouds object-management facilities (*i.e.*, the so-called "Clouds objects"). Invocations of operations on Clouds objects are handled by the compiler through operations on the Clouds object manager on the machine on which the invoking code is running. The Clouds object on which the operation is being invoked need not be located on the same machine as the invoking code; the object manager then makes a *remote procedure call* (RPC) to the object manager on the machine on which the called object resides. The location—local or remote—of the object being operated upon, however, need not concern the programmer, as the RPC process is transparent above the object-management level.

2.2.3 Support for Actions The action concept provides an abstraction of the idea of work in the Clouds system; an action represents a unit of work. Actions provide *failure atomicity*, that is, they display "all-or-nothing" behavior: an action either runs to completion and *commits* its results, or, if some failure prevents completion, it *aborts* and its effects are cancelled as if the action had never executed.

Support for actions in the Aeolus language is relatively low-level. At present, the methodology of programming with actions is not as well-understood as the methodology of programming with objects; thus, rather than providing high-level syntactical abstractions such as those available for object programming, Aeolus allows access to the full power and detail of the Clouds system facilities for action management. The major syntactic support provided by Aeolus for action programming is in the programming of *action events*, *recoverable areas*, *permanent* and *per-action variables*, and *action invocations*.

At several points during the execution of an action, the action interacts with the *action manager* of the Clouds system to manage the states of objects touched by that action, including writing those states to *permanent* (stable or safe) storage, and recovering previous permanent states upon failure of an action. Thus, failure atomicity may be provided by the action management system. The *action events* include:

event name	*purpose*
BOA	beginning of action
toplevel_precommit	prepare for commit of a toplevel action
nested_precommit	prepare for commit of a nested action
commit	normal end of action (EOA)
abort	abnormal end of action

The interactions with the Clouds action manager necessary when such events take place are done by default procedures supplied by the Aeolus compiler and runtime system; these procedures are

called *action event handlers*. When an action event occurs for a particular action, the action manager(s) involved invoke the event handlers for each object touched by that action.

As was described above, by use of the `autorecoverable` class of object, the programmer may take advantage of the recovery facilities of the Clouds system by having the compiler generate the necessary code automatically. This automatic recovery mechanism requires recovery of the entire state of the object, and uses the default action event handlers. However, it is sometimes possible for the programmer to improve the performance of object recovery by providing one or more object-specific event handlers which make use of the programmer's knowledge of the object's semantics; these programmer-supplied event handlers then replace the respective default event handlers for that object. Thus, if object class keyword `recoverable` is specified in the definition header of the object being implemented, the programmer may give an optional *action event part* in the object's implementation part. Following the keywords `action events`, the programmer lists the name of each action event handler provided by the object implementation as well as the name of the action event whose default handler the specified handler is to override. Thus, for example, the specification (in an object implementing a bounded-stack abstraction):

```
action events
   stack_BOA overrides BOA,
   stack_nested_precommit overrides nested_precommit
```

indicates that the default handlers for the `BOA` and `nested_precommit` action events are to be replaced by the procedures named `stack_BOA` and `stack_nested_precommit`, respectively, for the bounded-stack object type only.

As mentioned above, if an object being implemented is of class `recoverable`, then some of its variables may be declared in a `recoverable` area. When a nested action first invokes an operation on a recoverable object (''touches'' that object), the action is given a new *version* of the recoverable area which initially has the same value as the version belonging to the action's immediate ancestor. The set of versions belonging to uncommitted actions which have touched a recoverable object is maintained on a *version stack* by a Clouds action manager. When a nested action commits, its version replaces that of its immediate ancestor. When a toplevel action commits, its version is saved to permanent storage. If an action is aborted, its version is popped from the version stack. Thus, recoverable areas (in conjunction with appropriate use of synchronization) provide *view atomicity*, that is, an action does not see the intermediate (uncommitted) results of other actions. Also, the use of recoverable areas allows the programmer to provide finer granularity in the specification of that part of the object state which must be recoverable, since the use of automatic recovery on an object (the `autorecoverable` object class) requires recovery on the entire state of the object. The interaction with the action manager necessary to manage the states of recoverable areas is implemented by the action event handlers as described above. Again, the default event handlers may be overridden by programmer-supplied event handlers for the entire object to achieve better performance.

It may sometimes be desirable to make large data structures resilient. In such cases, the recoverable area mechanism may be inefficient, since it requires the creation of a new version of the entire recoverable area for each action which modifies the area. Often in such cases the programmer may take advantage of knowledge of the semantics of the data structure to efficiently program the recovery of the data structure. The Aeolus language provides two constructs which aid in the custom programming of data recovery, the so-called *permanent* and *per-action variables*, constructs proposed by McKendry [McKe85].

Any type may be given the attribute `permanent`. This attribute indicates that members of that type are to be allocated on the *permanent heap*, a dynamic storage area in the object storage of each object instance. This area receives special treatment by the Clouds storage manager; in particular, it is *shadow-paged* during the `toplevel_precommit` action event.

Aeolus also provides the per-action variable construct. A per-action variable specification resembles a recoverable area specification, and its semantics is also similar, in that each action which touches an object with per-action variables gets its own version of the variables; however, the programmer may access the per-action variables not only of the current action, but also of the parent of the current action. Also, per-action variables are allocated in *non-permanent storage*, that is, in storage the contents of which may be lost upon node failure. The variables in a per-action specification are accessed as if they were fields in a record described by the specification; two entities of this "record type" are implicitly declared: `Self` and `Parent`, which refer respectively to the per-action variables of the current action and its immediate ancestor.

Permanent and per-action variables may be used together to simulate the effect of recoverable areas at a much lower cost in space per action. In general, the per-action variables are used to propagate changes to the resilient data structure up the action tree; these changes are then applied during the `toplevel_precommit` action event to the actual data structure in permanent storage. The use of permanent and per-action variables is shown more fully in the Aeolus papers cited above.

The right-hand side of an assignment statement may take the form of an *action invocation*. Here, the right-hand side (which consists of an operation invocation which, if the operation is value-returning, is embedded in another assignment statement) is invoked as an action; the *action ID* of this action is assigned to the variable designated by the left-hand side of the action invocation. Thus, for example, if the bounded-stack object mentioned above were defined as a recoverable object, one might invoke one of its operations as an action:

```
aID := action( stack_instance @ push( elem ) )
```

The action ID may be used as a parameter in operations on the action manager which provide information about the status of the action, cause a process to wait on the completion of an action, or explicitly cause an action to commit or abort. By use of additional syntax not shown here, the programmer may specify that an action be created as a "top-level" action, that is, as an action with no ancestors; a top-level action cannot be affected by an abort of any other action. Otherwise, the action is created as a "nested" action, that is, as a child (in the so-called *action tree*) of the action which created it; as described below, a nested action may be affected by an abort of one of its ancestors. Optionally, a *timeout value* may be specified in the action invocation clause; if the action has not committed by the expiration of this timeout, the action will be aborted. If no timeout value is specified, a system-defined default value is used. The detailed semantics of action invocations, and requirements on objects that may have operations invoked as actions, are described in the papers on Aeolus cited above.

3. Overview of Distributed Locking

In this section, we outline a model of concurrency control and replication management for the Clouds system, called Distributed Locking (DL). The linguistic and runtime mechanisms required to support DL are described in the following sections.

In the DL methodology, derivation of a replicated object from its single-site implementation consists essentially of two steps:

1. The user writes a single-site definition and implementation of the object. This implementation includes specification of all lock types used by the object to ensure view atomicity in the presence of concurrently-executing actions.
2. The user writes an *availability specification* (`availspec`) for the object. This specifies the number of replicas of each instance of the object to be generated, the replication control policies to be used, and (optionally) the relative availabilities of the modes of each lock type specified by the object. If no `availspec` is provided, the object is assumed to be nonreplicated.

The `availspec` construct is discussed in detail in Section 4. Note that availabilities are expressed in terms of the modes of locks rather than in terms of operations. Together with the *domain* notion, with which lock granularities are expressed in Aeolus/Clouds, this gives the user more latitude in the expression of relative availabilities than is provided in related work (described in Section 6).

The automation of replication provided by the DL methodology is based on a concept similar to that of *action events* and *object events* as discussed in Section 2. The programmer may specify the interaction of an object with the action management system at critical points in the processing of an action via writing handlers for the action events; handlers for object events allow the object to participate in its creation and destruction. In a similar spirit, we have identified two critical points in the handling of an operation invocation on a replicated object: the *lock event*, during which the invocation attempts to synchronize some subset of the replicas of the object; and the *copy event*, during which the state resulting from the invocation is transmitted to the subset of replicas synchronized during the corresponding lock event. These events correspond to the concurrency control and consistency maintenance aspects of replication control, respectively. Note that the names we have chosen for these events reflect the lock-based synchronization and stable storage-based recovery mechanisms of Clouds; extensions to other synchronization and recovery methods are considered briefly in Section 7. For reasons examined in Section 5, we require that an invocation on a replicated object be made in the context of an action.

Policies for control of concurrency among replicas, and for control of the copying of state among replicas, are expressed in a `lock` object event handler and a `copy` action event handler, respectively, in the `availspec` for an object. Preprogrammed default handlers for these events, implementing commonly-used schemes such as quorum consensus, may be requested by the user if appropriate. If the user wishes to provide application-specific handlers for these events, the same system-provided primitives used in the construction of the default handlers are available for use in programming user-specified handlers. These primitives are described in Section 5, and example event handlers using the primitives are also presented there.

4. Availability Specifications

As discussed in Section 6, the Consensus Locking model of Herlihy allows the specification of the availability properties of an abstract data type in terms of the initial and final quorums required for an operation. It has already been mentioned that in the Distributed Locking model it makes sense to speak of the availability properties of lock modes (rather than of operations, as in other schemes). Some means is needed of allowing the programmer to specify these availability properties for an object without requiring modification of the single-copy version of the object definition or implementation.

In Distributed Locking as implemented in the Aeolus/Clouds system, the availability properties of a replicated object are specified in a separate compiland for that object type, called the *availability specification part* (or `availspec`, for short). The properties specified in an

`availspec` include the number of replicas, the replication management algorithm desired (*e.g.*, quorum assignment, available-copies, etc.), the name of each lock type declared by the implementation of that object along with the names of that lock's modes, and (optionally) the availability relationships among the modes of each lock type used by the implementation of that object. All internal and/or non-Clouds objects used by a replicated object must also have a replication specification; this requirement is applied recursively to these objects. The availability information of a non-Clouds object is inherited by the object which imports it; thus, the effect is as if locks declared by non-Clouds objects were instead declared by the importing Clouds object.

If a voting method is chosen, the quorum assignments for each lock may be derived from the replication specification using integer programming methods. The availability relationships among locking modes, expressed as relative availabilities, may be transformed into constraints on the space of feasible solutions; the objective function may be chosen to maximize the minimum availability over the locking modes subject to these constraints. The construction of this linear program is discussed in more detail later in this chapter. This information is transformed by the Aeolus compiler into a table of replication management information which is stored in the `TypeTemplate` of the Clouds object (the `TypeTemplate` is used by the Clouds system to generate instances of an object type). This information is placed in the header information of each object instance and is used by the Distributed Locking primitives to guide the selection of sets of replicas for Distributed Locking (see Section 5).

The Aeolus *availability specification* bears some resemblance to the *fault-tolerance specification* of the HOPS system (cf. Section 6). However, in HOPS the programmer must select among several predefined policies for replication control; there is no provision for user programming of these policies. The ability of the programmer to specify `lock` and `copy` event handlers as well as the provision of primitives in support of programming these handlers allows the use of a wider range of replication control policies with the Aeolus `availspec` construct.

4.1 Example of an Availability Specification

A sample `availspec` making use of the `quorum` event handlers is given in Figure 5. This `availspec` applies to a resilient symbol table object, the definition for which is presented in Appendix A; the implementation of this object is presented and discussed in detail in [Wilk87]. For the purposes of this example, we will describe only the locks declared in the symbol table implementation.

For synchronization purposes, a lock is declared which allows the entire symbol table to be locked:

```
symtable_lock : lock ( exact : [ exact ], nonexact : [ nonexact ] )
```

Note that operations acquiring `symtable_lock` in `exact` mode may run concurrently with other operations acquiring it in `exact` mode, and similarly for `nonexact` mode; however, operations attempting to acquire the lock in `exact` mode must block on those holding it in `nonexact` mode, and *vice versa*. The use of `symtable_lock` is to lock out changes during the `exact_list` operation. Thus, the `insert` and `delete` operations acquire this lock in `nonexact` mode, while the `exact_list` operation acquires it in `exact` mode; the `lookup` and `quick_list` operations need not acquire the lock at all.

The resilient `symtab` object must operate in an action environment; thus, additional synchronization is needed to assure the view atomicity of modifications. For this purpose, a new lock variable is introduced which controls the visibility of names in the symbol table:

```
availspec of object symtab ( d : unsigned ) is

   ! Availability specification of the symbol table object using
   ! the quorum consensus scheme.  The DistLock pseudo object
   ! definitions are imported automatically by all availspecs,
   ! but we must import the quorum definitions to use its
   ! predefined handlers.

   import quorum

   ! First, we specify the degree of replication (the number of
   ! replicas).  Here, the degree is taken from an additional
   ! parameter, d, which is specified during creation of an
   ! instance of this object.

   degree is d

   ! The resilient symtab object defines two locks, each with two
   ! modes.  We define the relative availabilities for the modes
   ! of each lock as follows.  The relative availabilities are
   ! used in the constraints of an integer program which is used
   ! in turn to generate the quorum assignments for each lock
   ! mode.

   lock symtable_lock with exact = nonexact

   lock name_lock with read > write

   ! The definitions of the lock and copy events.  Here, we just
   ! use the predefined handlers for quorum consensus.

   availspec events
      quorum_lock overrides lock_event,
      quorum_copy overrides copy_event

end availspec. ! symtab
```

Figure 5. Availability Specification for the Resilient Symbol Table

```
name_lock : lock ( write : []          ,
                   read  : [ read ] ) domain is name_type
```

This lock defines the usual multiple reader/single writer protocol over values of `name_type` (that is, the type of keys). The `insert` and `delete` operations acquire this lock in `write` mode; the `find` operation acquires it in `read` mode. Thus, attempts to insert or delete a given name may not execute concurrently with each other or with attempts to read that name.

The degree of replication (*i.e.*, the number of replicas for a given instance of `symtab`) is given as a formal parameter to the `availspec`; the actual parameter is supplied (in addition to any object parameters specified by the definition part of the object) during creation of object

instances.

The `availspec` also specifies the relative availabilities of the modes of each lock declared by `symtab`. Here, the two modes of `symtable_lock` are declared to have the same availability level; however, the `read` mode of `name_lock` is declared to be more available than the `write` mode. The relative availability declarations are used to determine the size of quorums for each mode.

Finally, the alternate handlers for the `lock` and `copy` events are specified. Here, the `quorum_lock` and `quorum_copy` operations made accessible by importing the `quorum` pseudo-object are used.

4.2 Computing Quorum Assignments

When a voting method is used for replication control, the system requires information about the minimum number of replicas required to constitute a quorum for each lock mode. As shown in the example `availspec` in the previous section, the programmer may specify the relative availabilities of the modes of each lock. This information is used to generate constraints for an integer program which computes the actual quorum requirements; the requirements for the modes of each lock of the object are then stored in the object state in an array associated with that lock. A primitive is provided for use in a `lock` event handler which returns the minimum quorum size associated with the lock and mode active at the invocation of the handler (that is, the request for which caused the `lock` event). The Distributed Locking primitives are described in Section 5.

The integer program used to generate the quorum information for each lock is built as follows. If the *i*th variable of the integer program represents the minimum number of replicas required to constitute a quorum for mode *i* of the lock, then the objective function is chosen to minimize the maximum value over all of the variables. As the availability of a mode is inversely proportional to the size of the quorum required for that mode, the objective function has the effect of maximizing the minimum availability over the modes. The relative availabilities of the locking modes as specified by the programmer in the `availspec` are used as constraints on the integer program; if no relative availabilities were specified, the availabilities of the modes are taken to be equal. There are additional constraints generated by the requirement of voting methods that the quorums of each pair of modes intersect (that is, that the sum of each pair of variables be greater than or equal to the degree of replication plus one), as well as that the value of each variable be nonnegative and be less than or equal to the degree of replication.

5. Support for Distributed Locking

As defined in Section 3, the term Distributed Locking refers to a *methodology* for deriving a replicated implementation from its single-copy version, as well as to a *mechanism* to support this methodology. A powerful feature of Distributed Locking is that it does not assume any particular *policy* for replication control. Although the user may easily specify use of one of several default policies in the areas of replica concurrency control and state copying, it also allows the user to explicitly program policies for these purposes. The mechanisms provided by Distributed Locking for support of both default and user-programmed policies are described below.

5.1 Naming Replicated Objects

The mechanism required for support of Distributed Locking requires modifications to the Clouds object naming scheme to support replication.

We have considered two different capability-based naming schemes which may be used in support of *state cloning*, as described in Section 1. The first scheme requires minimal changes to the Clouds kernel, but relies on facets of the Clouds object lookup mechanism which may not be applicable to other systems. In Clouds, the search for an object begins locally (that is, on the node which invoked the search), and—if the object is not found locally—proceeds to a broadcast search. If the internal objects belonging to a replica are constrained to reside on the same node as their parent object, then the local search will locate the local instance of the internal object. (This constraint is not considered to be onerous, since the internal objects of each replica need to be highly available to that replica in any case, and thus should logically reside on the same node as the parent replica.) Thus, each replica of an object (each of which resides on a separate node) may maintain its set of internal objects using the same capabilities as each other replica. (This situation may be created by initializing one replica, and then cloning its state to the other replicas.) Although there will thus be multiple instances (on separate nodes) of internal objects referenced by the same capability, there should be no problems caused by this, since—by the definition of internal object—only the parent object or its internal objects may possess the capability to an internal object, and the object search will always locate the correct (local) instance. Thus, state cloning may be used to copy the state of a replica to the other replicas without causing the problems with respect to internal objects described in Section 1 (concerning references to internal objects contained in the replica's state), since under this scheme all replicas may use the same capabilities for referencing internal objects. This scheme is an extension of a facility already supported by the Clouds kernel for cloning read-only objects such as code. This scheme is called *vertical replication*, since it maintains the grouping of internal objects with their parent object.

The other naming scheme makes fewer assumptions about the lookup mechanism than vertical replication, but requires more kernel modifications. In the second scheme, each instance of the replicas' internal objects is again named by the same capability, at least as far as the user is concerned; however, the kernel maintains several additional bits associated with each capability identifying a unique instance. (These additional bits may be derived, for example, from the birth node of the instance.) When a (parent) replica invokes an operation on an internal object, the kernel selects one of the replicas of the internal object according to some scheme (*e.g.*, iteration through the list of nodes containing such objects until an available copy is located). Thus, a set of replicas of internal objects is maintained in a "pool" for access by all parent replicas. Again, each parent appears to use the same (user) capability to reference a given internal object, so the problems of state cloning disappear. Since this scheme maintains a logical grouping of the copies of an internal object, rather than grouping internal objects with their parent object, this scheme is called *horizontal replication*. One such naming scheme is described in a paper by Ahamad *et al.* [Aham87]

The attractions of the vertical replication scheme are that it is conceptually simple, that it requires no modifications to the kernel capability-handling mechanisms, and that, by requiring coresidence, it enforces a property which enhances availability. To see this, recall that independent failure modes are desirable among different replicas of a replicated object, since the probability that the replicated object will be available is the probability that *any one* of the set of replicas will be available. On the other hand, dependent failure modes are desirable among a given replica and its internal objects, since the probability that the given replica will be available is the probability that *all* of the set of internal objects will be available. Requiring coresidence of objects related by logical nesting introduces dependence of their failure modes.

Unfortunately, the vertical replication scheme is not viable in general, since the coresidence requirement may sometimes be unrealistic. It may sometimes be the case that it is impossible to

satisfy coresidence, due to the size of nested objects (making it impossible to accommodate them on the same node), or due to insufficient space because of previously-existing objects on that node. Thus, vertical replication must be abandoned as lacking sufficient generality in its applicability. Fortunately, the horizontal replication scheme does not share this drawback.

The horizontal replication scheme has been further developed in a recent paper by other researchers on the Clouds project [Aham87a]. However, the invocation scheme may be altered to take advantage of coresidence when possible. The search scheme used for invocation of replicated objects in the paper cited above involves a random choice among the set of replicas. This differs markedly from the current Clouds search scheme for non-replicated objects, which is essentially as follows:

```
if <object found locally> then
   <perform invocation on local object>
else
   <perform global search>
end if
```

This search scheme may be modified to take advantage of coresidence as follows:

```
if <object found locally> then
   <perform invocation on local object>
else
   if <object is replicated> then
      <select randomly among the set of replicas>
   else
      <perform global search>
   end if
end if
```

Note that, if only one replica is stored per node, the local search involves only the so-called "user capability;" that is, it does not involve the extra bits used by the "kernel capability" to distinguish among replicas. If one allows more than one replica per node, some use of the kernel capability must be made to select an appropriate instance; this may require specific knowledge of which replicas are stored at which nodes.

5.2 Invocation of *Lock* and *Copy* Events

Support of the Distributed Locking mechanism requires modification of the Acolus/Clouds object and action management facilities in two areas.

1. When an operation attempts to obtain a lock on an instance of a replicated object, locks are obtained at some appropriate subset of its replicas, by invoking the *lock* event handler on that object. (Using terminology introduced by Ahamad and Dasgupta [Aham87a], the replica at which the original invocation took place is called the *primary cohort* [*p-cohort*]; the other members of the locked subset of replicas are called *secondary cohorts* [*s-cohorts*].)

2. During the handling of the precommit event of the controlling action, the state of each p-cohort touched by that action is copied to its s-cohorts, by invoking the *copy* event handler on each p-cohort.

In Section 1, two methods of copying object state applicable to the Clouds model were identified:

1. *idempotent execution*, or execution of an invocation at each member of the set of replicas; and

2. *cloning*, or execution of an invocation at a single replica, and then explicitly copying its state to the other replicas.

Because of the drawbacks of idemexecution (including the possibility of repeated invocations on objects external to the replicated object, as well as the difficulty of handling invocations with non-deterministic results in this scheme), the most viable mechanism seems to be cloning. However, the Distributed Locking mechanism does not preclude the use of idemexecution in the *copy* event, and provides primitives for its support.

Since a replicated object may have an arbitrary structure of logically nested objects, it is a non-trivial problem to determine exactly what state of which objects must be copied to implement a cloning operation. That is, it does not suffice to merely copy the state of the *p-cohort* to its *s-cohorts*; the states of all objects nested with respect to the *p-cohort* which were involved in the given operation must also be copied to their respective replicas (the nested objects of the *s-cohorts*). Fortunately, the Clouds action mechanism provides a means of determining which objects must be cloned: the action manager maintains a list of objects *touched* by an action. (This is the reason behind requiring that invocations on replicated objects take place in the context of an action.) Indeed, one need only perform cloning upon commit of an action, since the results of an action become visible to other actions only after commit. At that time, the so-called "shadow set" of each touched object is available. (In very simplified terms, this is the set of pages in the object's recoverable area which have been modified by the action.) If the constraint is made that all replicated objects be recoverable, then to implement cloning, one need only copy the shadow set of each touched object to the other replicas in that object's set, and perform the commit actions of storage management at each replica. The shadows are committed at each of the s-cohorts as if the shadows had been produced by execution at that s-cohort.

5.3 Primitives for *Lock* and *Copy* Event Handlers

If the user wishes to provide application-specific handlers for these events, the same system-provided primitives used in the construction of the default handlers are available for use in programming user-specified handlers. These primitives, and their purposes, include those for such purposes as:

- acquisition at a specific replica of the currently-requested lock (with the same mode and value, if any), for implementing lock propagation;
- invocation at a specific replica of the same operation (with the same parameters) requested at the current replica, for implementing idemexecution;
- broadcast of state shadow sets to all replicas holding a specified lock (with a specified mode and value), for implementing cloning via shadows; and
- invocation at a specific replica of an arbitrary operation, for implementing cloning via logs or state reconciliation strategies.

The intention is to provide facilities at a level sufficiently low to accommodate all schemes of interest. Some other useful predefined objects, such as those implementing list abstractions, are available for such purposes as maintaining and traversing the list of replicas at which locks have been obtained (and to which the object state must later be copied).

The primitives described above are encapsulated in an Aeolus pseudo-object called *DistLock*. The definition of *DistLock* is presented in its entirety in Appendix B.

```
implementation of pseudo object quorum is

   ! Here, we define handlers for the lock and copy events which
   ! implement quorum consensus.  This pseudo object is imported
   ! by any availspec wishing to use its predefined handlers.

import DistLock

procedure quorum_lock () is
   ! A simple-minded lock event handler for quorum consensus.
   ! Locks are obtained on at least a minimum quorum assignment
   ! specified by the assignment matrix generated by the
   ! importing availspec.

   this_version ,
   max_version  : version_number
   num_locked   ,
   good_replica : replica_number

   begin
      ! Find out how many replicas have been locked already by
      ! the current action.
      num_locked := DistLock @ currently_locked()

      ! Initially, the latest version seen is set to this
      ! instance's version number.
      max_version  := DistLock @ my_version()

      ! Attempt to lock all available replicas.
      for r in replica_number[ 1 .. DistLock @ degree() ] loop
         if DistLock @ lock_replica( r, this_version ) then
            num_locked += 1
            if this_version > max_version then
               max_version  := this_version
               good_replica := r
                  ! remember which replica has the latest version
            end if
         end if
      end loop

      ! At least a quorum of replicas must have been locked.  If
      ! not, abort the invoking action.
      if num_locked < DistLock @ quorum_size() then
         Abort_Myself()
      end if

      ! If there is a later version of the state than that of
      ! this replica, copy it here.  (This updates the local
      ! version number.)
      if good_replica <> DistLock @ my_replica() then
         if not DistLock @ get_state( good_replica ) then
            Abort_Myself()  ! replica was unavailable
```

```
        end if
      end if

      ! Copy the local state to all replicas which have version
      ! number less than that of the local copy.
      for r in replica_number[ 1 .. DistLock @ degree() ] loop
         if not DistLock @ send_state( r ) then
            Abort_Myself()  ! replica was unavailable
         end if
      end loop
   end procedure ! quorum_lock


procedure quorum_copy is
   ! The copy event handler for quorum consensus.  The shadow set
   ! is copied to the set of replicas locked in the lock event.

   begin
      if not DistLock @ broadcast_shadows() then
         Abort_Myself()  ! copy was unsuccessful
      end if
   end procedure ! quorum_copy

end implementation. ! quorum
```

Figure 6. Lock and Copy Event Handlers for Quorum Consensus

5.4 Examples of Event Handlers in Distributed Locking

A sample implementation of `lock` and `copy` event handlers using the General Quorum Consensus algorithm are given in Figure 6. The treatment of these event handlers has been kept on a fairly naive level to avoid obscuring neither the general lines of the algorithm used nor the use of the Distributed Locking primitives. The handlers are encapsulated in a pseudo-object called `quorum` which may be imported by an `availspec` in order to use its handlers.

As described in a previous section, the replica of an object at which an operation is invoked is called the *primary cohort* or *p-cohort*; a request for a lock at the p-cohort causes its lock event handler to be activated. The handler for the `lock` event, here called `quorum_lock`, attempts to lock each other available replica (called *secondary cohort* or *s-cohort*) by use of the `lock_replica` Distributed Locking primitive; if successful, this primitive returns the version number of the new s-cohort as an `out` parameter. The maximum version number over all s-cohorts is determined and compared with the version number of the p-cohort; if the latter is not the latest version, the state of the s-cohort having the latest version is copied to the p-cohort. In any case, at this point the latest state is copied to all s-cohorts having earlier states. If the number of s-cohorts is not at least as great as the quorum assignment for the requested lock mode, the enclosing action is aborted.

When the action enclosing the operation invocation prepares to commit, the `copy` event handler (here called `quorum_copy`) is activated. This handler uses the `broadcast_shadows` primitive to copy the shadow set (of changed pages) of the p-cohort to the s-cohorts locked in all activations of the `lock` event handler by the current action. If the copy is successful, the shadow sets are committed at the s-cohorts as well as the p-cohort to yield the updated state.

There are obvious improvements which might be made to this simple version of `quorum`. For example, `quorum_lock` relies on the `lock_replica` primitive to "fall through" when an attempt is made to lock a replica which is already an s-cohort. A more sophisticated implementation could maintain a set of replica numbers representing the current set of s-cohorts in order to avoid the overhead of a remote invocation for each redundant `lock_replica` call.

The use of the `broadcast_shadows` primitive in `quorum_copy` requires that the states of all s-cohorts be identical to that of the p-cohort when the `lock` event handling is complete, so that the shadow set broadcast during the `copy` event can be committed into a common permanent state at each replica; this is achieved by copying the state of the replica with the latest version number to those replicas with earlier versions of the state. This implementation assumes that it is uncommon for the version number of a replica to be "out of synch" with its fellow replicas, which is a reasonable assumption if most, if not all, replicas are available to become s-cohorts during each `lock` event. If this assumption is invalid, it may be more efficient to avoid copying of the latest state to the s-cohorts during the `lock` event *and* copying shadow sets during the `copy` event by copying the entire state of the p-cohort to the s-cohorts during the `copy` event.

6. Related Work

In this section, previous work on the properties of resilience and availability in distributed applications is examined. The issues of resilience in work related to Clouds have been examined in previous Clouds dissertations [Allc83a, Spaf86, Pitt86]; The discussion in this chapter thus concentrates on the issues of availability as treated by other researchers, except where the previous work relates to the linguistic support for resilience as provided in Aeolus.

The study of the use of replication to enhance availability first occurred in the area of distributed database systems, and was later adopted in the area of distributed operating systems. Thus, the problems in the control of concurrency among replicated objects were studied and, for the most part, solved by database researchers; the concurrency control methods used by the operating systems projects described below are largely derived from the database research. A survey of this work appears in a recent book by Bernstein *et al.* [Bern87] The history of these efforts is also summarized by Wright [Wrig84]. However, the research in database systems has been limited to consideration of "flat" objects, such as records or files; as was shown in Section 1, the generalization to arbitrary structure of objects in distributed operating systems research leads to problems related to the mechanisms used for the copying of state among replicas.

6.1 Replication in Database Systems

As with most of the topics involved in the study of distributed systems, the synchronization and recovery of replicated data was first studied in the area of distributed database systems. Examples of database concurrency control methods methods are voting schemes [Giff79, Thom79], available-copy methods [Good83], primary copy methods [Ston79], and token-passing schemes [LeLa78]. The intent of these methods is to ensure consistency of the replicated data by requiring access to a special copy or set of copies of the data during failures or partitions. Primary copy methods allow access to a copy during a network partition only if the partition possesses the designated primary copy of the data. Token-passing schemes are an extension of primary copy methods; a token is passed among sites holding a copy of data, and that copy at the site currently holding the token is considered the primary copy. Yet another extension of primary copy methods are the voting schemes. Each copy of the data object is assigned a (possibly different) number of votes; a partition possessing a majority of the votes for that object may access it.

Finally, available-copy methods follow a "read-one, write-all-available" discipline. A read operation may access any *initialized* copy (that is, one which has already processed a write operation). A write operation must access all copies; those which are unavailable for writing are called *missing writes*. A validation protocol, which runs after all reads and writes of a transaction have either been processed or timed out, guarantees one-copy serializability. This protocol ensures that all copies for which missing writes were recorded are still unavailable, and that all copies accessed are still available. Several researchers have recently proposed enhancements to the original available-copies algorithm [Skee85, El-A85, Long87].

El Abbadi has recently proposed a paradigm for developing and analyzing concurrency control protocols for replicated databases, especially those handling partition failures [El-A87]. He has also proposed a new protocol, developed within this paradigm, which allows read and write access to data despite partitions.

6.2 Replication in Operating Systems

Previous work in the area of replication of data in distributed operating systems includes the ISIS system at Cornell, the Eden system and the Emerald language at the University of Washington, the Argus system at MIT, Cooper's work on the Circus replicated procedure call facility at Berkeley, the HOPS project at Honeywell, Inc. and Herlihy's work at MIT (General Quorum Consensus) and CMU (Avalon).

6.2.1 ISIS The ISIS system developed at Cornell [Birm84, Birm85] supports *k-resilient* objects (objects replicated at $k+1$ sites and which can tolerate up to k failures) by means of checkpoints and the "available copies" algorithm. ISIS objects can refer to other objects, although apparently all such "nested" objects are considered to be external. This system provides both availability and *forward progress;* that is, even after up to k site failures, enough information is available (at the remaining sites possessing an object replica) that work started at the failed sites can continue at these remaining sites. This is accomplished through a *coordinator-cohort* scheme, where one replica acts as master during a transaction to coordinate updates at the other, "slave" replicas ("cohorts"). The choice of which replica acts as coordinator may differ from transaction to transaction. The object state is apparently copied from the coordinator to the cohorts via a cloning operation; this operation has been described as propagating a checkpoint of the entire coordinator [Birm84], or, in a more recent paper, as propagating the most recent version in a version stack [Birm85]. In the current system, it is assumed that the network is not subject to partitioning.

In ISIS, a transaction is not aborted when a machine on which its coordinator is running fails (transactions are usually aborted only when a deadlock situation arises). Rather, the transaction is resumed at a cohort from the latest checkpoint, in what is called *restart mode;* this cohort becomes the new coordinator. Operations which the coordinator had executed after the latest checkpoint took place must be re-executed at the new coordinator.

In the course of an operation on a k-resilient object, the coordinator may perform operations on other objects to which it contains references. Such operations on "nested" objects are called *external actions*. Inconsistencies can arise due to external actions performed during restart mode; operations performed on external objects by the new coordinator in this mode were also performed by the old coordinator before it failed. Thus, unless the operations on external objects are idempotent, inconsistencies can arise. (This problem is closely related to the problem of idemexecution on external objects, discussed in Section 1.) This problem is solved in ISIS by requiring external objects to retain results of operations; these retained results are associated with a transaction ID. When a new coordinator takes over from a failed coordinator and enters restart mode, it uses the same IDs for its external operations, and rather than re-execute these operations,

the external objects merely return the associated results.

There is also an idemexecution scheme due to Joseph [Jose85, Jose86] which was apparently implemented as an experiment using the ISIS system as a testbed, rather than as part of the ISIS replication mechanism itself. In Joseph's scheme, the coordinator performs the requested operation, and then instructs its cohorts to perform the same operation.

Recently, a new version of the ISIS system, called ISIS-2, has been designed; it is anticipated that this new system will be operational by Fall 1987. The ISIS-2 design exploits a new abstraction called the *virtually synchronous process group* [Birm87]. In this abstraction, a distributed set of processes cooperate to perform work in an environment in which broadcasts, failures, and recoveries are made to appear synchronous.

6.2.2 Eden and Emerald The Eden system [Alme83, Blac86] was under development at the University of Washington from September 1980 until late 1986; the system has been operational on a collection of VAX systems (and later Sun workstations) since April 1983. Support in the Eden system for replication has been studied at both the kernel level and the object level. The kernel level implementation of replication support is called the *Replect* approach (for replicated "Ejects," or Eden objects), while the object level implementation is called *R2D2* (for "Replicated Resource Distributed Database"). Both implementations use quorum consensus for concurrency control.

In Eden, objects are *active*, that is, each object encapsulates—besides data and operations on the data—one or more active processes which are permanently associated with that object. Normally, an object has two forms: an *active form* (AF) which exists in volatile memory, and a *passive representation* (PR), which is a checkpoint of the AF on disk. The PRs are maintained in *permanent object databases* (PODs), one of which exists on each node in the system. In the unreplicated case, an object has only one AF and one PR at any time.

In the Replect approach [Prou85], although the PR of the object is replicated, the object still has only one AF at any time. Thus, one capability is used to refer to all replicas of the object. Hence, a Replect is referenced by the user in the same way as a normal object; the Replect mechanism is transparent to clients of a Replect. A transaction management facility is required to ensure that multiple AFs are not produced by competing transactions despite crashes. (The basic Eden system does not provide a transaction management facility.) Updates are performed by selecting one of the PODs to act as transaction manager in a master/slave protocol.

In the R2D2 approach [Pu85], each replica is a complete object, consisting of an AF as well as a PR. Each replica is unaware of the others, but clients must refer to the replicated object by using a set of capabilities (to the multiple AFs), one for each replica; thus, this mechanism is less transparent than the Replect approach. R2D2 objects are stored in a replicated hierarchical directory structure. Invocations on objects replicated using R2D2 must use a specialized transaction manager (called R2D2TM), which traverses the replicated directory and handles the multiple updates on members of the set of replicas. Members of the set which are unavailable due to crashes are replaced via *regeneration*. The level of the directory in which the unavailable object is maintained must be updated to reflect the replacement.

The basic Eden system was not designed to handle partitions [Noe85]. The two replication approaches described above compensate for this lack in differing degrees. Using the R2D2 approach, an object will be able to regenerate if its partition contains a copy of the PR and a suitable number of machines, and will then be able to continue to operate. However, upon the resolution of the partition, the states of competing versions of the object must be merged. Thus, the Eden authors prefer to use voting methods, allowing simple merging of partitions, although

replicas in a partition without a majority will be unable to operate. Using the Replect approach, on the other hand, problems arise even with voting methods due to the problem of avoiding having multiple AFs. If a partition contains a quorum of PRs, but the AF is gone, it is not possible to tell if the AF is inactive (dead) or in another partition. If one is to allow multiple AFs, a state-merging scheme must also be provided, since the isolated AF may be updated with no attempt to checkpoint to the PRs.

No mention is made in the Eden references of support for arbitrary structure of objects or of the associated problems of state propagation.

Another project at the University of Washington is concerned with the design and implementation of an object-oriented language for distributed applications [Blac86a, Blac87]. Emerald provides a hierarchy of object classifications similar to that provided by Aeolus (as described in Section 2); however, selection of an appropriate classification for an object is made automatically by the Emerald system. Emerald does not at present provide support for fault tolerance.

6.2.3 Argus The Argus system at MIT [Lisk83, Lisk84, Lisk83a, Weih83] is a language and system for distributed applications which has evolved from the CLU language. Argus provides an object construct (called *Guardian*) which encapsulates data and processes, giving an abstraction of a physical node or server. Argus also retains the *cluster* construct from CLU, which provides functionality similar to that of local objects in Aeolus; however, the syntax of Guardians is not similar to that of clusters. Resilience in Argus is based on the notion of system-provided primitive *atomic data types*, from which user-defined atomic data types may be constructed. These primitive atomic data types also define the synchronization properties of the user-constructed types. Experience with programming a distributed, collaborative editing system in Argus has been described by Greif *et al.* [Grei86]; one criticism arising out of this experience was that they were sometimes forced to use a Guardian where a cluster might have been more appropriate.

Recent work at MIT has been concerned with availability issues in distributed services [Lisk86, Lisk87]. The researchers have developed a method for constructing highly-available services which maintain a form of view atomicity despite the presence of old information in their states. This method requires that the properties of the information be *stable* in the sense that once a property becomes true, it does not change thereafter. Availability methods possessing this property are useful in applications such as distributed garbage collection.

6.2.4 Circus Cooper has investigated a mechanism called the *replicated procedure call*, which he implemented at Berkeley in a system called *Circus* [Coop84, Coop85]. In Cooper's scheme, although replicas of an object have no knowledge of each other, they are bound (via run-time support) into a server called a *troupe* which may be accessed by client objects. (The client objects know that the server is replicated.) An object in Circus may have arbitrary structure, containing references to both internal and external objects. However, the object is currently required to be deterministic. His scheme uses idemexecution for state propagation. When a troupe accesses an external troupe (a so-called "many-to-many" call), results of operations on objects of the server troupe are retained by the callees; these results are associated with *call sequence numbers*, and are returned when subsequent calls by the replicas of the caller troupe with the same sequence numbers are encountered, thus avoiding the inconsistencies possible with idemexecution on external objects. Concurrency control is by majority voting. Thus, if a partition does not have a majority of troupe members, invocations will not be able to proceed.

6.2.5 The HOPS Project The *Honeywell Object Programming System* (HOPS) [Hone86] under development at Honeywell, Inc., has research goals similar to those of our methodology research. The stated goals of the HOPS project are:

- to alleviate what is seen as a lack of experience in the field of distributed systems in implementing mechanisms which perform failure detection, failure recovery, and resource reconfiguration;
- to provide programming support for developing fault-tolerant distributed applications; and
- to assess the actual benefits and costs of such mechanisms in terms of performance, reliability, and availability.

HOPS consists of an implementation language derived from Modula-2 together with a distributed runtime support system. The language requires that HOPS objects (or *HOPjects*) be specified in three parts: an interface specification, a body (or implementation specification), and a *fault tolerance specification*. In the latter, the programmer may specify attributes and policies relating to recovery, concurrency control, and replication which are to be used for that object, thus giving the programmer a choice among several mechanisms provided by HOPS in each of these areas. The distributed runtime system (together with the underlying host operating system) provides facilities for naming and addressing objects, communication, failure detection and recovery, local and distributed transaction management, concurrency control, recovery, and replication. HOPS is currently being implemented on a network of Sun-3 workstations under the Sun version of Unix 4.2.

Mechanisms for achieving fault tolerance in HOPS include the *distributed recovery block* (DRB) mechanism and *distributed conversations*. (The recovery block and conversation mechanisms are described in detail in a book by Anderson and Lee [Ande81] as well as in the HOPS report cited above.) Basically, the combination of the DRB and conversation mechanisms provide fault tolerance by what is essentially "software modular redundancy." Processes at two or more nodes execute one of a set of differing sections of code (called *try blocks*) which implement the same specified function; the results of these try blocks must pass the same *acceptance test* (possibly with majority voting), or the participating processes are rolled back to a checkpoint (called a *recovery line*) and retry the computation with their alternate try blocks. Thus, both fail-stop and some Byzantine-style failures may be detected and tolerated by this scheme.

6.2.6 General Quorum Consensus and Avalon Herlihy's work on General Quorum Consensus [Herl84] concerns the extension of quorum intersection methods to take advantage of the semantic properties of abstract data types. Previously, work on quorum methods—mostly in the database area—has been limited to a simple read/write model of operations. Herlihy's extensions allow the selection of optimal quorums for each operation of an abstract data type based on the semantics of that operation and its interaction with the other operations of the data type.

Herlihy's method is based on the analysis of the algebraic structure of abstract data types. This entails the construction of a "quorum intersection graph," each node of which represents an operation of the data type, and each edge of which is directed from the node representing an operation *O1* to the node representing operation *O2*, where each quorum of *O2* is required to intersect each quorum of *O1*. From the quorum intersection graph, optimal quorums for each operation may be calculated, given the number of replicas of the data, and the desired availability of each operation in relation to the other operations of the data type.

Herlihy shows that his method can enhance the concurrency of operations on replicated data over that obtained from a read/write model of operations. He also claims advantages for his methods in the support of on-the-fly reconfiguration of replicated data, and in enhancing the availability of

the data in the presence of network partitions.

More recently, Herlihy has developed two new methods for integrating concurrency control and recovery for abstract data types, called *Consensus Locking* and *Consensus Scheduling*. In these schemes, Herlihy requires that the quorum intersection relation and the lock conflict relation (the complement of the lock compatibility relation) for an object satisfy a common *serial dependency relation* on that object; he notes that, in practice, the lock conflict and quorum dependency relations will be the same [Herl85]. A detailed comparison of Consensus Locking is presented in [Wilk87].

A third scheme, called *Layered Consensus Locking*, extends the Consensus Locking method by associating a *level* with each activity in the system [Herl85a]. Activities at a higher level are serialized after activities at a lower level. If an activity executing at a given level is unable to make progress after a failure with its current quorum assignment, it may restart at a higher level and switch to another quorum assignment. Each initial quorum for an invocation at level *n* is required to intersect with each final quorum for an event at levels <= *n*.

Herlihy and Wing recently have been developing a set of linguistic features, called *Avalon*, for support of transaction processing [Herl87]. Avalon is intended to be implemented as extensions to pre-existing languages such as Ada and C++, and is built on the *Camelot* distributed system developed at CMU. Avalon provides support for action event handling resembling that provided by Aeolus, as described in Section 2. Avalon also provides support for testing serialization orders dynamically.

7. Conclusions and Future Directions

In this paper, methods of achieving resilience and availability in the Clouds system have been examined. In the course of this work, we have designed a systems programming language providing access to the Clouds features of objects and actions, features which—used in conjunction with the Aeolus runtime support—provide powerful support for resilience of data and computations. Although automatic support for resilient objects—the paradigm provided by other systems with goals similar to those of Clouds—is provided as an option in Aeolus, facilities are also provided that allow the programmer to specify resilience mechanisms more appropriate to the semantics of the object when desirable.

We have taken a similar approach in designing a scheme, Distributed Locking, for supporting high availability of Clouds objects. Most distributed system projects providing support for replication assume a certain policy for replication control, usually quorum consensus. In the course of recent research, several algorithms for replication control displaying availability properties more desirable than those of other algorithms in some situations have been proposed. Thus, it seemed advisable to provide the capability of supporting several different policies for replication control rather than assuming any one policy. Predefined policies may be accessed as defaults if the programmer so desires; however, since a replication control scheme other than one of those foreseen as a pre-programmed policy may prove more appropriate to the semantics of a given object, our scheme also allows the programmer to develop new policies using the same library of support primitives used to develop the default policies.

7.1 Performance of Distributed Locking

We consider the Distributed Locking mechanism in the form described in this paper to be a tool for research into replication techniques rather than a production system for real-world applications. However, it may be instructive to estimate the performance of the mechanism in a sample application in order to demonstrate how such estimates may be derived in other cases;

these derivations would be useful primarily for comparison of different replication techniques. As a sample application, we assume a replicated object of degree three, each replica having a permanent storage area consisting of ten pages, and using the quorum consensus handlers (as described in Section 5) for replication control. For simplicity, we also assume that the action being performed on the replicated object consists of an operation invocation that does not visit other objects, and that this operation causes the entire permanent storage area to be shadowed (the worst case).

The two-phase commit protocol in Kenley's action management design [Kenl86] requires a total of four message/acknowledgment pairs per site visited by an action. In Phase I (the Prepare phase), the coordinating site must send each visited site a *prepare* message; if all goes well, each visited site responds with a *prepared* message indicating success. In Phase II (the Completion phase), if all visited sites have responded positively to the prepare message, the coordinator sends a *commit* message to each visited site; if commit is successful, each visited site responds with a *committed* message.

In the Clouds prototype, a message/acknowledgement pair for a message of maximum size 1.5 Kbytes requires approximately thirty milliseconds [Stri88]. (The network driver has not yet been examined for possible performance improvements.) Thus, the messaging overhead of an action commit is 120 ms per site, to which must be added 60 ms for the action manager to write a commit log at the coordinating site.

Timings for writing to stable storage in the Clouds prototype have been measured by Pitts [Pitt86]. To install the shadow version, there is a constant overhead of approximately 120 ms; there is also a cost per page of the shadow set which ranges between 25 ms (if the write is sequential and does not require a seek) and 51 ms (if the write is random). (These figures are based on a driver for a relatively small, slow disk; a driver for a much faster disk has recently been developed, and should yield much better performance figures, perhaps one-third or better of those of the slow disk.)

If a communications environment is assumed that does not allow broadcast, then messages must be sent separately by action management to each replica of an object touched by an action to perform a commit. If the quorum consensus protocol is used, separate messages must also be sent to each replica to transmit the shadow set of the coordinating site (*p-cohort* in the terminology of Section 5) to the other replicas (*s-cohorts*); a maximum of three 512-byte pages may be transmitted per message. However, stable storage processing may be done concurrently at each replica once the shadow set is transmitted. Let R represent the degree of replication of the invoked object, and P be the number of pages of permanent storage in the object. Then, for a non-broadcast environment, the overhead of the quorum consensus copy event, *i.e.*, the time required to commit the simple action invocation described above on a replicated object, is given by:

$$120R+60 + 30\lceil P/3\rceil (R-1) + 51P+120$$

where the first term represents the contribution by action management overhead, the second term the time required to transmit the shadow set to the s-cohorts (excluding the p-cohort), and the third term the time required to write the shadow set to stable storage at each replica (assuming the worst case in which all writes are random); all constants are in milliseconds. For the sample application described above (where R=3 and P=10), a commit of the simple action would require approximately 1290 ms.

In the Clouds prototype, the action management messages as well as the shadow sets may be broadcast to the replicas, thus eliminating the need for sequential messages to each replica. In this environment, the overhead of the copy event reduces to:

$$120+60+30\lceil P/3\rceil+51P+120$$

and the copy event of the sample application would require approximately 870 ms in the worst case. (If the estimates given above for the performance of the faster disk are assumed, the overhead becomes approximately 510 ms.) Note that this expression does not depend R, the number of replicas. The expression is indeed close to the overhead involved in committing a single-site object on a different site than the coordinator for the action; the expression for the single-site case does not include the second term (the overhead of broadcasting the shadow set to the s-cohorts). The time to commit the single-site object is thus approximately 750 ms for the slow disk in the worst case. If all writes on the slow disk were sequential (perhaps a more normal case), the overheads would be 610 ms for the replicated object vs. 490 ms for the single-site object.

A similar analysis may be performed of the additional activity required during the lock event handling. Considering the handler for quorum consensus, the worst case occurs when all replicas are available to be locked (requiring R-1 messages to perform the locking), and when the latest version must be copied to all s-cohorts (requiring $\lceil P/3\rceil$ (R-1) messages). The overhead for the sample application in this worst case would thus be approximately 300 ms. If the state is up-to-date at all cohorts at the time of the lock event, however, the overhead would reduce to just that involved for the locking messages, in this case 60 ms.

7.2 Current Status

The first prototype of the Clouds operating system has been implemented and is operational. This version is referred to Clouds v.1. This is being used as an experimental testbed by the implementors. Results of performance tests with this prototype are available in other publications on Clouds [Spaf86, Pitt86], and are summarized in [Dasg87]. The experience with this version has taught us that the approach is viable. It also taught us how to do it better.

The lessons learned from this implementation are being used to redesign the kernel and build a new prototype. The basic system paradigm, the semantics of objects, and the goals of the project remain unchanged and v.2. will be identical to v.1. in this respect.

The structure of Clouds v.2. is different. The operating system will consist of a minimal kernel called *Ra*. Ra will support the basic function of the system, that is location independent object invocation. The operating system will be built on top of the Ra kernel using system level objects to provide systems services (user object management, synchronization, naming, atomicity and so on).

The Ra implementation is now in progress. The action management subsystem, the design of which is described in Kenley's thesis [Kenl86], is being redesigned to work with Ra. A compiler and runtime system for the Aeolus language have been implemented in a Pascal variant (with some C and assembler in the runtime system); the compiler is being rewritten in Aeolus for portability purposes. Implementation of Distributed Locking will be possible once the redesigned action management subsystem is in place, as the interfaces to action management from Aeolus already exist.

7.3 Future Directions

The version of the Distributed Locking scheme described in this paper is based on the policies of lock-based synchronization and stable storage-based recovery, implemented by the action management and storage management subsystems of Clouds, respectively. As mentioned in Section 2, the Clouds kernel is designed so that these subsystems may be replaced with others implementing different policies. We are currently considering the effects on the DL mechanism of the replacement of locks with timestamp-based synchronization, and the replacement of shadowed stable storage with log-based recovery. We anticipate that these changes will require additions to the library of primitives supporting DL.

In addition, we are considering the effects on DL of relaxing the fail-stop assumption. This will require primitives supporting the reconciliation of replica states which have diverged via operating in separate partitions. These primitives may be used in conjunction with the *reinitialization* object event described in Section 2.

8. Acknowledgements

We would like to express our appreciation to our co-workers on the Clouds project, past and present, on whose work and ideas we have built. We would especially like to thank David Pitts for his comments on earlier drafts of this paper.

This work was funded in part by NSF grant DCR-8405020, by NASA grant NAG-1-430, and by RADC Contract Number F30602-86-C-0032.

REFERENCES

[Aham87] Ahamad, M., P. Dasgupta, R. J. LeBlanc, and C. T. Wilkes. "Fault-Tolerant Computing in Object Based Distributed Operating Systems." *PROCEEDINGS OF THE SIXTH SYMPOSIUM ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS* (IEEE Computer Society), Williamsburg, VA (March 1987): 115-125.

[Aham87a] Ahamad, M., and P. Dasgupta. "Parallel Execution Threads: An Approach to Fault-Tolerant Actions." TECHNICAL REPORT GIT-ICS-87/16, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, March 1987.

[Allc82] Allchin, J. E., and M. S. McKendry. "Object-Based Synchronization and Recovery." TECHNICAL REPORT GIT-ICS-82/15, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1982.

[Allc83] Allchin, J. E., and M. S. McKendry. "Synchronization and Recovery of Actions." *PROCEEDINGS OF THE SECOND SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING* (ACM SIGACT/SIGOPS), Montreal (August 1983).

[Allc83a] Allchin, J. E. "An Architecture for Reliable Decentralized Systems." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1983. (Also released as technical report GIT-ICS-83/23.)

[Alme83] Almes, G. T., A. P. Black, E. D. Lazowska, and J. D. Noe. "The Eden System: A Technical Review." TECHNICAL REPORT 83-10-05, University of Washington Department of Computer Science, October 1983.

[Andc81] Anderson, T., and P. A. Lee. *Fault Tolerance, Principles and Practice*. Englewood Cliffs, NJ: Prentice-Hall International, 1981.

[Bern87] Bernstein, P. A., V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley, 1987.

[Birm84] Birman, K. P., T. A. Joseph, T. Raeuchle, and A. El-Abbadi. "Implementing Fault-Tolerant Distributed Objects." *PROCEEDINGS OF THE FOURTH SYMPOSIUM ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS*, Silver Spring, MD (October 1984): 124-133.

[Birm85] Birman, K. P. "Replication and Fault-Tolerance in the ISIS System." *PROCEEDINGS OF THE TENTH SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES* (ACM SIGOPS), Orcas Island, Washington (December 1985). (Also released as technical report TR 85-668.)

[Birm87] Birman, K. P., and T. A. Joseph. "Exploiting Virtual Synchrony in Distributed Systems." TECHNICAL REPORT TR 87-811, Department of Computer Science, Cornell University, Ithaca, NY, February 1987.

[Blac86a] Black, A., N. Hutchinson, E. Jul, and H. Levy. "Object Structure in the Emerald System." TECHNICAL REPORT 86-04-03, Department of Computer Science, University of Washington, Seattle, WA, April 1986.

[Blac87] Black, A., N. Hutchinson, E. Jul, H. Levy, and L. Carter. "Distribution and Abstract Types in Emerald." *TRANSACTIONS ON SOFTWARE ENGINEERING* (IEEE)

13, no. 1 (January 1987). (Also available as University of Washington Technical Report 85-08-05.)

[Blac86] Black, A. P., E. D. Lazowska, J. D. Noe, and J. Sanislo. "The Eden Project: A Final Report." TECHNICAL REPORT 86-11-01, Department of Computer Science, University of Washington, Seattle, WA, 1986.

[Coop84] Cooper, E. "Circus: A Replicated Procedure Call Facility." *PROCEEDINGS OF THE FOURTH SYMPOSIUM ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS*, Silver Spring, MD (October 1984): 11-24.

[Coop85] Cooper, E. "Replicated Distributed Programs." *PROCEEDINGS OF THE TENTH SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES* (ACM SIGOPS), Orcas Island, WA (December 1985): 63-78. (Available as *Operating Systems Review* **19**, no. 5.)

[Dasg87] Dasgupta, P., R. LeBlanc, and W. Appelbe. "The Clouds Distributed Operating System: Functional Description, Implementation Details, and Related Work." TECHNICAL REPORT GIT-ICS-87-28, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, July 1987. (To appear in the Proceedings of the Eighth International Conference on Distributed Computing Systems..)

[El-A85] El-Abbadi, A., D. Skeen, and F. Cristian. "An Efficient, Fault-Tolerant Protocol for Replicated Data Management." *PROCEEDINGS OF THE FOURTH SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS* (ACM SIGACT-SIGMOD) (March 1985).

[El-A87] El-Abbadi, A. "A Paradigm for Concurrency Control Protocols." PH.D. DISS., Department of Computer Science, Cornell University, Ithaca, NY, 1987.

[Giff79] Gifford, D. K. "Weighted Voting for Replicated Data." *PROCEEDINGS OF THE SEVENTH SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES* (ACM SIGOPS), Pacific Grove, CA (December 1979).

[Good83] Goodman, N., D. Skeen, A. Chan, U. Dayal, R. Fox, and D. Ries. "A Recovery Algorithm for a Distributed Database System." *PROCEEDINGS OF THE SECOND SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS* (ACM SIGACT-SIGMOD), Atlanta, GA (March 1983).

[Grei86] Greif, I., R. Seliger, and W. Weihl. "Atomic Data Abstractions in a Distributed Collaborative Editing System." *CONFERENCE RECORD OF THE THIRTEENTH SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES* (ACM SIGACT/SIGPLAN), St. Petersburg Beach, FL (January 1986). (Extended Abstract.)

[Herl84] Herlihy, M. "Replication Methods for Abstract Data Types." PH.D. DISS., Laboratory for Computer Science, Massachussetts Institute of Technology, Cambridge, MA, May 1984. (Also released as Technical Report MIT/LCS/TR-319.)

[Herl85] Herlihy, M. "Atomicity vs. Availability: Concurrency Control for Replicated Data." TECHNICAL REPORT CMU-CS-85-108, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, February 1985.

[Herl85a] Herlihy, M. "Using Type Information to Enhance the Availability of Partitioned Data." TECHNICAL REPORT CMU-CS-85-119, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, April 1985.

[Herl87] Herlihy, M. P., and J. M. Wing. "Avalon: Language Support for Reliable Distributed Systems." *PROCEEDINGS OF THE SEVENTEENTH INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING*, Pittsburgh, PA (July 1987). (Also available as Technical Report CMU-CS-86-167.)

[Hone86] Honeywell, Inc. "Fault Tolerant Distributed Systems." INTERIM SCIENTIFIC REPORT, Computer Sciences Center, Honeywell Inc., Golden Valley, MN, November 1986. (RADC Contract No. F30602-85-C-0300.)

[Jose85] Joseph, T. A. "Low-Cost Management of Replicated Data." PH.D. DISS., Department of Computer Science, Cornell University, Ithaca, NY, November 1985. (Also released as Technical Report TR 85-712.)

[Jose86] Joseph, T. A., and K. P. Birman. "Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems." *TRANSACTIONS ON COMPUTER SYSTEMS* (ACM) 4, no. 1 (Febuary 1986): 54-70.

[Kenl86] Kenley, G. G. "An Action Management System for a Distributed Operating System." M.S. THESIS, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as technical report GIT-ICS-86/01.)

[LeBl85] LeBlanc, R. J., and C. T. Wilkes. "Systems Programming with Objects and Actions." *PROCEEDINGS OF THE FIFTH INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS*, Denver (July 1985). (Also released, in expanded form, as technical report GIT-ICS-85/03.)

[LeLa78] LeLann, G. "Algorithms for Distributed Data-Sharing Systems Which Use Tickets." *PROCEEDINGS OF THE THIRD BERKELEY WORKSHOP ON DISTRIBUTED DATA MANAGEMENT AND COMPUTER NETWORKS*, Berkeley, CA (August 1978).

[Lisk83] Liskov, B., M. Herlihy, P. Johnson, G. Leavens, R. Scheifler, and W. Weihl. "Preliminary Argus Reference Manual." PROGRAMMING METHODOLOGY GROUP MEMO 39, Laboratory for Computer Science, Massachussetts Institute of Technology, Cambridge, MA, October 1983.

[Lisk83a] Liskov, B., and R. Scheifler. "Guardians and Actions: Linguistic Support for Robust Distributed Programs." *TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS* (ACM) 5, no. 3 (July 1983).

[Lisk84] Liskov, B. "Overview of the Argus Language and System." PROGRAMMING METHODOLOGY GROUP MEMO 40, Laboratory for Computer Science, Massachussetts Institute of Technology, Cambridge, MA, February 1984.

[Lisk86] Liskov, B., and R. Ladin. "Highly-Available Distributed Services and Fault-Tolerant Distributed Garbage Collection." PROGRAMMING METHODOLOGY GROUP MEMO 48, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1986.

[Lisk87] Liskov, B. "Highly-Available Distributed Services." PROGRAMMING METHODOLOGY GROUP MEMO 52, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, February 1987.

[Long87] Long, D. D. E., and J.-F. Paris. "On Improving the Availability of Replicated Files." *PROCEEDINGS OF THE SIXTH SYMPOSIUM ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS* (IEEE Computer Society), Williamsburg, VA

(March 1987): 77-83.

[McKe85] McKendry, M. S. "Ordering Actions for Visibility." *TRANSACTIONS ON SOFTWARE ENGINEERING* (IEEE) **11**, no. 6 (June 1985). (Also released as technical report GIT-ICS-84/05.)

[Noe85] Noe, J. D., A. B. Proudfoot, and C. Pu. "Replication in Distributed Systems: The Eden Experience." TECHNICAL REPORT TR-85-08-06, Department of Computer Science, University of Washington, Seattle, WA, September 1985.

[Pitt86] Pitts, D. V. "Storage Management for a Reliable Decentralized Operating System." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as Technical Report GIT-ICS-86/21.)

[Pitt87] Pitts, D. V., and P. Dasgupta. "Object Memory and Storage Management in the Clouds Kernel." TECHNICAL REPORT GIT-ICS-87/15, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, March 1987. (To appear in the Proceedings of the Eighth International Conference on Distributed Computing Systems.)

[Prou85] Proudfoot, A. B. "Replects: Data Replication in the Eden System." M.S. THESIS, Department of Computer Science, University of Washington, Seattle, WA, December 1985. (Also released as University of Washington Technical Report TR-85-12-04.)

[Pu85] Pu, C., J. Noe, and A. Proudfoot. "Regeneration of Replicated Objects: A Technique for Increased Availability." TECHNICAL REPORT TR-85-04-02, Department of Computer Science, University of Washington, Seattle, WA, April 1985.

[Schl83] Schlichting, R. D., and F. B. Schneider. "An Approach to Designing Fault-Tolerant Computing Systems." *TRANSACTIONS ON COMPUTER SYSTEMS* (ACM) **1**, no. 3 (August 1983): 222-238.

[Skee85] Skeen, D. "Determining the Last Process to Fail." *TRANSACTIONS ON COMPUTER SYSTEMS* (ACM) **3**, no. 1 (February 1985): 15-30.

[Spaf86] Spafford, E. H. "Kernel Structures for a Distributed Operating System." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as technical report GIT-ICS-86/16.)

[Ston79] Stonebreaker, M. "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES." *TRANSACTIONS ON SOFTWARE ENGINEERING* (IEEE) **5**, no. 3 (May 1979).

[Stri88] Strickland, H. "Networking Support for a Distributed Operating System." M.S. THESIS, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1988. (In progress.)

[Thom79] Thomas, R. H. "A Majority Consensus Approach to Concurrency Control for Multiple-Copy Databases." *TRANSACTIONS ON DATABASE SYSTEMS* (ACM) **4**, no. 2 (June 1979).

[Weih83] Weihl, W., and B. Liskov. "Specification and Implementation of Resilient

Atomic Data Types." *SYMPOSIUM ON PROGRAMMING LANGUAGE ISSUES IN SOFTWARE SYSTEMS* (June 1983).

[Wilk85] Wilkes, C. T. "Preliminary Aeolus Reference Manual." TECHNICAL REPORT GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985. (Last Revision: 17 March 1986.)

[Wilk86] Wilkes, C. T., and R. J. LeBlanc. "Rationale for the Design of Aeolus: A Systems Programming Language for an Action/Object System." *PROCEEDINGS OF THE 1986 INTERNATIONAL CONFERENCE ON COMPUTER LANGUAGES* (IEEE Computer Society), Miami, FL (October 1986): 107-122. (Also available as Technical Report GIT-ICS-86/12.)

[Wilk87] Wilkes, C. T. "Programming Methodologies for Resilience and Availability." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987. (Also available as Technical Report GIT-ICS-87/32.)

[Wrig84] Wright, D. D. "Managing Distributed Databases in Partitioned Networks." PH.D. DISS., Department of Computer Science, Cornell University, Ithaca, NY, January 1984. (Also available as Cornell University Technical Report 83-572.)

Appendix A

In this appendix, the Aeolus definition part for a resilient symbol table object is presented. This example is used in Section 4.

```
definition of local object symtab
      ( name_type : type,  value_type : type ) is
   ! Single-copy symbol table object using the Aeolus/Clouds lock
   ! mechanisms for synchronization.  The definition part
   ! contains specifications of public constants, types, and
   ! operations defined by this object.  When compiled, it
   ! produces a symbol table file which may be imported by other
   ! objects using this object in their implementations.

operations

   procedure insert ( name  : name_type   ,
                      value : value_type  ,
                      error : out boolean ) modifies
      ! The insert operation places an entry into the
      ! symbol table.  error is set if an entry with the
      ! same name already exists.

   procedure delete ( name  : name_type   ,
                      error : out boolean ) modifies
      ! If the delete operation finds an entry with the
      ! given name, it removes the entry from the symbol table
      ! and frees its storage space.

   procedure lookup ( name  : name_type   ,
                      error : out boolean )
                    returns value_type examines
      ! The lookup operation tries to locate the entry with
      ! the given name and returns its value if it succeeds.
      ! error is set if the entry is not in the table.

   procedure quick_list () examines
      ! The quick_list operation provides a quick (dirty)
      ! listing of all names currently in the symbol table.

   procedure exact_list () examines
      ! The exact_list operation provides a listing of the
      ! exact state of the symbol table at a given point in time.
      ! To do this, it locks the whole symbol table, thereby
      ! excluding any changes during preparation of the listing.
      ! Thus, although exact_list, lookup, and
      ! quick_list operations may execute concurrently, and
      ! insert and delete operations which access
      ! different hash buckets may also execute concurrently,
      ! insert and delete operations must block on
      ! exact_list operations.

end definition.
```

Appendix B

In this appendix, the Aeolus definition part serving as the user interface to the Distributed Locking primitives is presented. This interface is discussed in Section 5.

```
definition of pseudo object DistLock is

    ! Interfaces to primitives provided for support of the
    ! Distributed Locking mechanism.  This pseudo-object is
    ! imported automatically by every availspec, and is not
    ! available for use by other compilands.

type replica_number is new unsigned
    ! A replica_number is used to name an individual replica of a
    ! group.  The naming scheme used here is the ''horizontal''
    ! method as described in Chapter VII of this dissertation.
    ! The replica_number is concatenated by the system to the
    ! capability of the object to which the invoking availspec
    ! belongs to form an extended capability as defined by the
    ! horizontal scheme.

type version_number is new longuns
    ! A version_number is used to compare the currency of
    ! the states of replicas.  The version number of an object is
    ! incremented whenever an invocation is performed on it, or
    ! when the state of the objected is updated by use of one of
    ! the designated operations described below.

operations

    procedure lock_replica ( rep : replica_number     ,
                             ver : out version_number )
                returns boolean modifies
        ! The lock_replica operation obtains the
        ! currently-requested lock at the replica denoted by rep.
        ! This operation should be invoked only within a lock event
        ! handler.  The lock variable, domain value, and mode
        ! requested are obtained from the context of the lock
        ! event which caused the invocation of the handler.
        ! The replica denoted by rep is added to a list of the
        ! replicas touched by the current action.
        ! The version number of the state of rep is returned
        ! in the out parameter ver.
        ! If lock_replica is unable to obtain the lock on
        ! rep, or if the requested lock is already held
        ! at rep by the current action, the operation returns
        ! FALSE, otherwise TRUE.
```

```
procedure invoke_replica ( rep : replica_number )
          returns boolean modifies
    ! The invoke_replica operation causes the current operation
    ! to be executed at the replica denoted by rep.  This
    ! operation should be invoked only within a copy event
    ! handler.  The operation number and other parameters are
    ! obtained from the context of the lock which caused the
    ! invocation of the handler.  The version number of rep
    ! is set to the value of that of the invoking object.
    ! This operation is used for implementing state copying by
    ! idemexecution.  If the invocation on rep is
    ! unsuccessful, the operation returns FALSE, otherwise
    ! TRUE.

procedure broadcast_shadows () returns boolean modifies
    ! The broadcast_shadow operation causes the ''shadow set''
    ! of the permanent state of the current action to be
    ! broadcast to all replicas at which locks were obtained by
    ! the current action via the lock_replica operation.
    ! The version numbers of the locked replicas are updated
    ! to equal that of the invoking object.  This
    ! operation should be invoked only within a copy event
    ! handler.  This operation is used for implementing state
    ! copying by cloning using shadows.  If all locked
    ! replicas successfully receive the shadow set, the
    ! operation returns TRUE, otherwise FALSE.

procedure get_state ( rep : replica_number )
          returns boolean modifies
    ! The get_state operation causes the state of the
    ! replica denoted by rep to be transmitted to the
    ! current object.  The state is installed at the current
    ! object, and its version number set to that of rep.
    ! If the transmission or installation fails, the operation
    ! returns FALSE, otherwise TRUE.

procedure send_state ( rep : replica_number )
          returns boolean modifies
    ! The send_state operation causes the state of the
    ! current object to be transmitted to the replica denoted
    ! by rep.  The state is installed at rep, and
    ! its version number set to that of the current object.  If
    ! the transmission or installation fails, the operation
    ! returns FALSE, otherwise TRUE.
```

```
   procedure invoke_acceptor ( rep   : replica_number ,
                               state : address        ,
                               len   : longuns        ) modifies
      ! The invoke_acceptor operation causes the invocation
      ! of the accept event handler at the replica denoted by
      ! rep.  The information the address of which is given by
      ! state and which is of length len bytes is copied to the
      ! environment of the accept handler at rep.  This operation
      ! may be used in a copy event handler to implement state
      ! copying by cloning using logs, or in a reinit event
      ! handler to implement state reconciliation strategies.

   procedure degree () returns replica_number examines
      ! The degree operation returns the total number of
      ! replicas of the current object including itself.

   procedure my_replica () returns replica_number examines
      ! The my_replica operation returns the replica number of
      ! the current object.

   procedure my_version () returns version_number examines
      ! The my_version operation returns the version number of
      ! the current object's state.

   procedure quorum_size () returns replica_number examines
      ! The quorum_size operation returns the minimum size of
      ! a quorum for the currently-requested lock mode.

   procedure currently_locked () returns replica_number
      ! The currently_locked operation returns the number of
      ! replicas on which the currently-requested lock mode has
      ! been obtained, including the current object.

end definition. ! DistLock
```

Appendix C

The Clouds Distributed Operating System: †

Functional Description, Implementation Details
and
Related Work.

Partha Dasgupta, Richard J. LeBlanc Jr., & William F. Appelbe.

School of Information and Computer Science
Georgia Institute of Technology
Atlanta GA 30332

Abstract

Clouds is an operating system in a novel class of distributed operating systems providing the integration, reliability and structure that makes a distributed system usable. *Clouds* is designed to run on a set of general purpose computers that are connected via a medium-to-high speed local area network. The structure of *Clouds* promotes transparency, support for advanced programming paradigms, and integration of resource management, as well as a fair degree of autonomy at each site.

The system structuring paradigm chosen for the *Clouds* operating system, after substantial research, is an object/thread model. All instances of services, programs and data in *Clouds* are are encapsulated in objects. The concept of persistent objects does away with the need for file systems, and replaces it with a more powerful concept, namely the object system. The facilities in *Clouds* include integration of resources through location transparency; support for various types of atomic operations, including conventional transactions; advanced support for achieving fault tolerance, and provisions for dynamic reconfiguration.

1. Introduction

Clouds is a distributed operating system under development. The goal of the *Clouds* project is to develop an instance of a class of distributed operating systems that provide the integration, reliability and structure necessary to make distributed computing system usable.

Clouds is designed to run on a set of general purpose computers (uniprocessors or multiprocessors) that are connected via a medium-to-high speed local area network. The major design objectives for *Clouds* are:

- Integration of resources through cooperation and location transparency.
- Usable support for various forms of atomicity, including transaction processing, and the ability to achieve fault tolerance (if needed).
- Efficient design and implementation.
- Simple and uniform interfaces for distributed processing.

The paradigm used for defining and implementing the software structure of the *Clouds* system, chosen after substantial research is an object/thread model. This model provides threads to support computation and objects to support an abstraction of storage. (These concepts are defined in sections 2 though 4). This model has been augmented to support atomicity of computation to provide support for reliable programs [Al83, ChDa87]. In this paper, we provide a functional description of the system (sections 2 to 6), some implementational details (section 7), and discussion of related work (section 9).

1.1. Current Status

The first version of the *Clouds* operating system has been implemented and is operational. This version is referred to Clouds v.1. This is being used as an experimental testbed by the implementors.

Some of the performance figures for *Clouds* v.1. were:

Local Invocations	10 msec
Remote Invocations	40 msec
Commit of 1 page data	180 msec

These figures are large due to several factors. The VAX architecture was not very suitable for implementing objects, and flushing of the translation buffers for each invocation causes the local invocation to be more expensive than expected [RaKh88]. The Ethernet hardware used in our VAX-11/750 is slow, and coupled with a non-optimized driver gives us poor performance on round trip messages and hence large remote invocation times. The disk used in the commit tests was also exceedingly slow (40msec seek, 25msec/page write.) However, the experience with this version has taught us that the approach works. It also taught us how to do it better.

The lessons learned from this implementation are being used to redesign the kernel and build a new version. The basic system paradigm, the semantics of objects and threads and the goals of the project remain unchanged and v.2. will be identical to v.1. in this respect.

The structure of *Clouds* v.2. is different. The operating system will consist of a minimal kernel called "*Ra*". Ra will support the basic function of the system, that is location independent object invocation. The operating system will be built on top of the Ra kernel using system level objects to provide systems services (user object management, synchronization, naming, atomicity and so on.)

† This research was partially supported by NASA under contract number NAG-1-430 and NSF under contract number DCS-8316590 and CCR-8619886.

2. Objects

All data, programs, devices and resources on *Clouds* are encapsulated in entities called objects. The only entity recognized by the system, other than an object is a thread. A *Clouds* object, at the lowest level of conception, is a virtual address space. Unlike virtual address spaces in conventional operating systems, a *Clouds* object is neither tied to any process nor is volatile. A *Clouds* object exists forever (like a file) unless explicitly deleted. As will be obvious in the following description of objects, *Clouds* objects are somewhat 'heavyweight', that is they are suited for storage and execution of large-grained data and programs. This is due to the fact that invocation and storage of objects bear some non-trivial overhead.

Every *Clouds* object is named. The name of an object, also known as its *capability*, is unique over the entire distributed system and does not include the location of the object. That is, the capability-based naming scheme in *Clouds* creates a uniform, flat system name space for objects, and allow for object mobility needed for load balancing and reconfiguration.

An object consists of a named address space, and the contents of the address space. Since it does not contain a process, it is completely passive. Hence, unlike objects in some object based systems, a *Clouds* object is not associated with any server process. (The first system to use passive objects, though in a multiprocessor system was Hydra [Wu74, WuLe81]).

Threads are the active entities in the system, and are used to execute the code in an object (details in sections 2 and 3). A thread executes in an object by entering it through one of several entry points, and after the execution is complete the thread leaves the object. Several threads can simultaneously enter an object and execute concurrently (or in parallel, if the host machine is a multiprocessor.)

Objects have structure. They contain, minimally, a code segment, a data segment and a mechanism for extending limits of storage allocated to the object. Threads that enter an object execute in the code segment. The data segment is accessible by the code in the code segment, but not by any other object. Thus the object has a wall around it which has some well-defined gateways, through which activity can come in. Data cannot be transmitted in or out of the object freely, but can be moved as parameters to the code segment entry points (see discussion on threads).

Clouds objects can be defined by the user or defined by the system. Most objects are user-defined. Some examples of system-defined objects are device drivers, name-service handlers, communication systems, systems software, utilities, and so on. The basic kernel (*Ra*) is not an object; it is an entity that provides the support for object invocation. A complete *Clouds* object can contain user-defined code and data; system-defined code and data that handle synchronization, recovery and commit; a volatile heap for temporary memory allocation; a permanent heap for allocating memory that will remain permanent as a part of the data structures in the object; locks; and capabilities to other objects.

Files in conventional systems can be conceived of a special case of a *Clouds* object. Thus, *Clouds* need not support a file system, but uses an object system. This is discussed in further detail in section 4.

Though, *Clouds* objects can be created, deleted and manipulated individually, the operating system is designed to support a class and instantiation mechanism. An object in the system can be an instance of its *template*. An object of a certain type is created by invoking a 'create' operation on the template of this type. Each template is created by invoking a create operation on a single template-template, which can create any template, if provided, as argument, the code and data definitions of the template. The templates, the template-template and all the instances thereof, are regular *Clouds* objects, and, as discussed earlier, they exist from the time of creation, until explicitly deleted.

3. Threads

The only form of activity in the *Clouds* system is the thread. A thread can be viewed as a thread of control that executes code in objects, traversing objects as it executes. Threads can span objects, and can span machine boundaries. In fact, machine boundaries are invisible to the thread (and hence to the user). Threads are implemented in the *Clouds* system as lightweight processes, comprising of a PCB and a stack (but no virtual space). A thread that spans machine boundaries is implemented by several processes, one per site.

Upon creation, a thread starts up at an entry point of an object. As the thread executes, it executes code inside an object and manipulates the data inside this object. The code in the object can contain a procedure call to an operation of another object. When a thread executes this call, it temporarily leaves the caller object and enters the called object, and commences execution there. The thread returns to the caller object after the execution in the called object terminates. The calls to the entry point of objects are called *object invocations*. Object invocations can be nested. The code that is accessible by each entry point is known as an *operation* of the object.

A thread executes by processing operations defined inside many objects. Unlike processes in conventional operating systems, the thread often cross boundaries of virtual address spaces. Addressing in an address space is, however, limited to that address space, and thus the thread cannot access any data outside an address space. Control transfer between address spaces occurs though object invocation, and data transfer between address spaces occurs through parameters to object invocation.

When a thread executing in an object (or address space) executes a call to another object, it can provide the called operation with arguments. When the called operation terminates, it can send back result arguments. That is, object invocations may carry parameters in either direction.

These arguments are strictly data, they may not be addresses. Note that names (capabilities) are data. This restriction is necessary as the address space of each object are disjoint, and an addresses is meaningful only in the context of the appropriate object. Parameter passing uses the copy-in-copy-out method.

4. The Object/Thread Paradigm

The structure created by a system composed of objects and threads has several interesting properties.

First, all inter-object interfaces are procedural. Object invocations are equivalent to procedure calls on modules not sharing global data. The modules are permanent. The procedure calls work across machine boundaries. (Since the objects exists in a global name space, there is no user-level concept of machine boundaries.) Although local invocations and remote invocations (also known as remote procedure calls or RPC) are differentiated by the operating system, this is transparent to the applications and systems programmers.

Second, the storage mechanism used in the object-based world is quite different from that used in the conventional operating systems. Conventionally, the file is the storage medium of choice for data that has to persist, especially since memory is tied to processes and processes can die and lose all the contents of their memory. However, memory is easier to manage, more suited for structuring data and essential for processing. The object concept merges these two views of storage, and creates the permanent virtual space.

For instance, a conventional file is a special case of an object. That is, a file is an object with operations such as read, write, seek, and so on, defined in it. These operations transport data in and out of the object through parameters provided to the calls.

Though files can be implemented using objects, the need for having files disappear in most situations. Programs do not need to store data in file-like entities, since they can keep the data in the data space in each object, structure appropriately. The need for user-level naming of files transforms to the need for user-level naming for objects.

Just as *Clouds* does not have files, it does not provide user-level support for file (or disk) I/O. In fact there is no concept of a "disk" or such I/O devices (except user terminals). The system creates the illusion of a huge virtual memory space that is permanent (non-volatile), and thus the need for using disk storage from a programmer's point of view, is eliminated.

Messages are a paradigm of choice in message-based distributed systems. In the object-thread paradigm, like the need for I/O, the need for messages is eliminated. Threads need not communicate through messages. Thus ports are not supported. This allows a simplified system management strategy as the system does not have to maintain linkage information between threads and ports.

Just as files can be simulated for those in need for them, messages and ports can be easily simulated by an object consisting of a bounded buffer that implements the send and receive operations on the buffer. However, we feel that the need for files and messages are the product of the programming paradigms designed for systems supporting these features, and these are not necessary structuring tools for programming environments.

A programmer's view of the computing environment created by *Clouds* is apparent. It is a simple world of named address spaces (or objects). These object live in computing systems on a LAN, but the machine boundaries are made transparent, creating a unified object space. Activity is provided by threads moving around amongst the population of objects through invocation; and data flow is implemented by parameter passing. The system thus looks like a set of *permanent address spaces* which support control flow through them, constituting what we term *object memory*.

This view of a distributed system does have some pitfalls. However these problems can be dealt with using simple techniques (implemented by the system), which are outlined below.

Threads aborting due to errors will leave permanent faulty data in objects they have modified. Failure of computers will result in similar mishaps. Multiple threads invoking the same object will cause errors due to race conditions and conflicts. More involved consistency violations may be the results of non-serializable executions. In a large distributed system, having thousands of objects and dozens of machines, corruption due to failure cannot be tolerated or easily repaired. The prevention of such situations is achieved through the use of atomicity at the processing level (not necessarily atomic actions). The following section gives a brief overview of the atomicity properties supported by *Clouds*.

5. Atomicity

The action support is an area where the *Clouds* v.1. and *Clouds* v.2. differ.

5.1. Actions in Clouds v.1.

In the first design, *Clouds* supported atomic actions and nested actions somewhat based on the model defined by Moss in his thesis [Mo81]. *Clouds* v.1. extended Moss's model by allowing custom tailored synchronization and recovery, as well as interactions between actions and non-actions.

The synchronization and recovery properties can be localized in objects, on a per object basis. The synchronization and recovery can be handled by the system (to adhere to Moss's semantics) or can be tailored by the user and thus provide facilities beyond those allowed by standard nested transactions. Customization is allowed by labeling of objects as "auto-sync" or "custom-sync" and "auto-recoverable" and "custom-recoverable". Further details can be found in [Wi87].

5.2. Atomicity in Clouds v.2.

The support for atomicity in *Clouds* v.2. has is roots in the above scheme, but has been changed in some respects. The following is a brief outline of the scheme. The actual methods used are discussed in greater detail in [ChDa87].

Instead of mandating customization of synchronization and recovery for application that cannot use strict atomicity semantics, the new scheme support a variety of *consistency preserving* mechanisms. The threads that execute are are of two kinds, namely *s-thread* (or *standard* threads) and *cp-threads* (or consistency-preserving threads). The s-threads have a "best effort" execution scheme and are not provided with any system-level locking or recovery. The cp-threads on the other hand are supported by locking and recovery schemes, provided by the system. When a cp-thread executes, all pages it reads are read-locked and the pages it updates are write-locked. The updated pages are written using a 2-phase commit mechanism when the cp-thread completes.

The data in the system has an *instantaneous* version and a *stable* version. In fact, if nested threads are used, the data has a stack of versions, the top being the instantaneous version and the bottom being the stable version. All the threads work on the instantaneous version. The data updated by cp-threads are committed when the cp-thread exits, while the data touched by the s-threads are committed "eventually", using a best effort semantics.

The cp-threads are allowed to interleave with s-threads, and also the cp-threads can be used to provide heavyweight as well as lightweight atomicity, using *gcp* and *lcp* operations, described below.

All threads are s-threads when created. The handling of cp-threads are programmed by the following scheme. All operations in objects in *Clouds* are tagged with a consistency label, the labels used are:

- Globally-Consistent (gcp)
- Locally-Consistent (lcp)
- Standard (s)
- Inherited (i)

An object can have any number of different labels on the operations. Also the same operation may have multiple entry points, labeled at different atomicity levels.

A s-thread executing a gcp or lcp operation converts to a cp-thread. A thread entering a lcp entry point, commits its updates (inside this object) as soon as it exits the object. This provides intra-object consistency rather than the inter-object consistency provided by the gcp operations, and thus is a cheap method of updating one object atomically. Locking and recovery are automatic.

The standard entry points do not support any locking or recovery. They can make use of "best-effort" semantics. They can also be used for non-traditional purposes such as peeking at incomplete results of actions (as they are not hindered by locking and visibility rules of actions). Locks are available for synchronizing non-actions, but recovery is not supported.

The other labels as well as combination of these labels in the same object (or in the same thread) lead to many interesting (as well as dangerous) variations. The complete discussion of the semantics as well as the implementation is beyond the scope of this paper, and the reader is referred to [ChDa87]

6. Programming Support

Systems and application programming for *Clouds* involves programming objects that implement the desired functionality. These objects can be expressed in any programming language. The compiler (or the linker) for the language, however, must be modified to generate the stubs for the various entry points, invocation handler, system call interfaces and the inclusion of default systems function handling code (such as synchronization and recovery.)

The language Acolus has been designed to integrate the full set of powerful features that the *Clouds* kernel supports. Aeolus currently supports the features of *Clouds* v.1. but is being expanded for added functionality of Ra and *Clouds* v.2. [LeWi85, Wi85, WiLe86].

Aeolus is the first generation language for *Clouds*. It does not support some of the features found in object-oriented programming systems such as inheritance and subclassing. Providing support for these features at the language level is currently under consideration.

7. Implementation Notes

The implementation of the *Clouds* operating systems has been based on the following guidelines:

- The implementation of the system should be suitable for general purpose computers, connected through popular networking hardware. Heterogeneous machines, though not crucial, should be allowed.
- Since the *Clouds* functionality is largely based on object invocation, support for objects should be efficient in order to make the system usable. Also, the naming, synchronization and recovery systems should be implementable with minimal overhead.
- Since one of the primary aims of *Clouds* is to provide the substrate for reliable, fault tolerant computing, the kernel and the operating system should provide adequate support for implementing fault tolerance. (Fault tolerance in not discussed in this paper, the reader is referred to [AhDa87].)
- The system design should be simple to comprehend and implement.

7.1. Hardware Configuration

Clouds v.1. was built on a three VAX-11/750 computers, connected through an Ethernet, equipped with RL02 and RA81 disk drives. The user interface was through the Ethernet, accessible from any Unix machine.

Clouds v.2. will be implemented on a set of Sun-3 class machines. The cluster of Clouds machines will be on an Ethernet, and user will be able access them through workstations running *Clouds* as well as any Unix workstation.

7.2. Software Configuration and Kernel Structure

The kernel (version 2.) used to support *Clouds* is called *Ra*. *Ra* is a native kernel running on bare hardware. The kernel is implemented in C for portability, and because the availability of C source for the UNIX kernel simplified the task of developing hardware interfaces such as device drivers.

The kernel runs on the native machine and not on top of any conventional operating system for two reasons. Firstly, this approach is efficient. As *Clouds* does not use much of the functionality of conventional operating systems (such as file systems), building *Clouds* on top of a Unix-like kernel make poor use of the host operating system. Secondly, the paradigms and the support for synchronization, recovery, shared memory and so on; used in *Clouds* are considerably different from the functionality provided by conventional operating systems, and major changes would be necessary at the kernel level of any operating system in order to implement *Clouds*.

The Ra kernel provides support for partitions, segments, virtual spaces, processes and threads. These are the basic building blocks for Clouds. The partitions provide non-volatile storage, the segments provide memory storage, which are used to build objects, which in turn reside in virtual spaces. Processes provide activity which are used to compose threads. A description of the design of Ra can be found in [BeHuKh87]

7.3. Object Naming and Invocation

The two basic activities inside the Ra kernel are system call handling and object invocations. System call handling is done locally, as in any operating system. The system calls supported by the Ra kernel include object invocation, memory allocation, process control and synchronization, and other localized systems functions. Object invocation is a service provided by the kernel for user threads. The attributes that object invocation satisfy are:

- Location independence.
- Fast, for both local and remote invocations.
- Failed machines should not hamper availability of objects on working sites, from working sites.
- Moving objects between sites, reassigning disk units and so on should be simple (for reconfiguration and fault tolerance support).

Location independence is achieved through a capability based naming system. Availability is obtained through decentralization of directory information and a search-and-invoke strategy coupled with a multicast based object location scheme, designed for efficiency [AhAm87]. Speed is achieved by implementing the invocation handlers at the lowest level of the kernel, on the native machine.

7.4. Storage Mangement

The storage management system handles the function required to provide the reliable, permanent object address spaces. As mentioned earlier, unlike conventional systems, where virtual address spaces are volatile and short-lived, *Clouds* virtual spaces contain objects and are permanent and long lived. The first version of the implementation is detailed in [Pi86].

The storage management system stores the object representations on disk, as an image of the object space. When an object is invoked, the object is demand paged into its virtual space as and when necessary. As the invocation updates the object, the updated pages do not replace the original copy, but have shadow copies on the disk. The permanent copy is updated only when a commit operation is performed on the object. The storage manager provides the support to commit an object using the two-phase commit protocol.

7.5. User Interfaces

User interfaces can make or break an operating system. Users do not like to switch systems, and have to re-learn the interfaces. We plan to use Unix and X-windows as our interface to *Clouds*. Unix programs can make use of *Clouds* facilities through invocation support provided by a *Clouds* library on Unix. Also, *Clouds* utilities will be available under X-windows. This will have several implications:

Firstly, *Clouds* can be treated as a back-end system to the Unix workstation, for distributed processing, computations, object-oriented programs and atomic programs. All these facilities will be available to Unix programs and the user.

Second, the user can access *Clouds* utilities through the X-window system, and thus making the learning time much smaller. We believe this approach will make *Clouds* easier to access and use, and we hope to build a large user community that is essential to the success of new operating systems.

8. Comparisons with Related Systems

Clouds is one of the several research projects that are building object-based distributed environments. Although there are differences between all the approaches, we feel that the area of distributed operating systems is not mature enough to conclusively argue the superiority of one approach over the other. In the following paragraphs we document the major differences between *Clouds* and some of the better known projects in distributed systems. (This list is not exhaustive).

One of the major difference between *Clouds* and some of the systems mentioned below is in the implementation of the kernel. Many systems implement the kernel as a Unix process[†], while *Clouds* is implemented as a native operating system (as are Mach and Alpha). *Clouds* is not intended to be an enhancement, or replacement of, the UNIX kernel. Instead, *Clouds* provides a different paradigm from that supported by UNIX (e.g., the UNIX paradigms of 'devices as files', unstructured files, volatile address spaces, pipes, redirection etc.)

8.1. Argus

Argus is a language for describing objects, actions and processes using the concept of a guardian. The language defines a distributed system to be a set of guardians, each containing a set of handlers. Guardians are logical sites, and each guardian is located at one site, though a site may contain several guardians. The handlers are operations that can access data stored in the guardian. The data types in Argus can be defined to be atomic, and atomic data types changed by actions are updated atomically when the action terminates [WeLi83, LiSc83]. The support for Argus is built on top of Unix, and provides all the facilities of the Argus language [Li87].

Some of the similarities between Argus and *Clouds* are in the semantics of nested actions. Both use the nested action semantics and locking semantics that are derived from Moss. This includes conditional commit and lock inheritance. However the consistency preserving mechanisms in *Clouds* have moved away from Moss's action semantics, substantially, though retaining the nested action semantics as a subset. Also the guardians and handlers in Argus have somewhat more than cosmetic similarities to objects in *Clouds*, as the design of *Clouds* was influenced by Argus.

The differences include the implementation strategies, programming support and support for reliability. The scheme of permanent virtual spaces provided by passive objects is a major difference. As mentioned earlier, Argus is implemented on top of a modified Unix environment. This is one of the reasons for the somewhat marginal performance of the Argus system observed in [GrSeWe86]. The programming support provided by Argus is for the Argus language. *Clouds* on the other hand is a general purpose operating system, not tied to any language. Though Aeolus is the preferred language at present, we have used C extensively for object programming. We have plans to implement more object-oriented languages for the the *Clouds* system.

8.2. Eden

Eden is a object-based distributed system, implemented on the Unix operating system at the University of Washington. Eden objects (called Ejects) use the active object paradigm, that is each object consists of a process and an address space. An invocation of the object consists of sending a message to the (server) process in the object, which executes the requested routine, and returns the results in a reply [Alm83, AlBl83, NoPr85].

Since every object in the system needs to have a process servicing it, this could lead to too many processes. Thus Eden has an *active* and a *passive* representation of objects. The passive representation is the core image of the object stored on the disk. When an object is invoked, it must be active, thus invoking a passive object involves activating it. A process is created by 'exec'-ing the core image of the object (frozen earlier), and then performs the required operation. The activation of passive objects is an expensive operation. Also concurrent invocations of objects are difficult and are handled through multithreaded processes or coroutines.

[†]The term *kernel* has been used quite frequently to describe the core service center of a system. However when this service is provided by a Unix process rather than a resident, interrupt driven monitor, the usage of the term is somewhat counter-intuitive.

The active object paradigm and the Unix-based implementation are some of the major differences between Eden and *Clouds*. Eden also provides support for transaction and replication objects (called Replects). The transaction support and replication were added after the basic Eden system was designed and have some limitations due to manner Unix handles disk I/O.

8.3. Cronus

Cronus is an operating system designed and implemented at BBN Laboratories. Some of the salient points of Cronus are the intergration of Cronus functions with Unix functions, the ability of Cronus to handle a wide variety of hardware and the coexistence of Cronus on a distributed set of machines running Unix, as well as several other host operating systems [BeRe85, GuDe86, ScTh86].

Like Eden, Cronus uses the active objects. This is necessary to be able to make Cronus run on top of most host operating systems. Cronus objects are handled by managers. Often a single manager can handle several objects, by mapping the objects into its address space. The managers are servers and receive invocation requests through catalogued ports. Any Unix process on any machine on the network can avail of Cronus services from any manager, by sending a message to the appropriate manager. By use of canonical data forms, the machine dependencies of data representations are made transparent. Irrespective of the machine types, any Unix machine can invoke Cronus objects in a location independent fashion.

8.4. ISIS

ISIS (version 1) is a distributed operating system, developed at Cornell University, to support fault tolerant computing. ISIS has been implemented on top of Unix. It uses replication and checkpointing to achieve failure resilience. If data object is declared to be k-resilient, the system creates k+1 copies of the object. The replicated object invocation is handled by invoking one replica and transmitting the state updates to all replicas. Checkpointing at each invocation is used to recover from failures [Bi85].

The goals and attributes of ISIS are different from Clouds. ISIS is built on top of some interesting communication primitives and is not built as a general purpose computing environment.

8.5. ArchOS and Alpha

Alpha is the kernel for the ArchOS operating system developed by the Archons project at Carnegie Mellon University. Like *Clouds*, the Alpha kernel is a native operating system kernel designed to run on the special hardware called Alpha-nodes. The Alpha kernel uses passive objects residing in their own virtual spaces, similar to *Clouds*. ArchOS is designed for real time applications supporting specialized defense related systems and applications [Je85, No87].

The key design criteria for ArchOS and Alpha are time critical computations and rather than reliability. Fault tolerance is handled to an extent using communication protocols. Real time scheduling has been a major research topic at the Archons project.

8.6. V-System

The V operating system has been developed at Stanford University. V is a compromise between message-based systems and object-based systems. The basic core of V provides light-weight processes and a fast communications (message) system. V message semantics are similar to object invocations in the sense that the messages are synchronous and use the send/reply paradigm. The relationship between processes conforms to the client-server paradigm. A client sends a request to the server, and the client blocks until the server replies [Ch88].

V allows multiple processes to reside in the same address space. Data sharing is through message passing, though shared memory can be implemented through servers managing bounded buffers. The design goals of V are primarily speed and simplicity. V does not provide transaction and replication support. These can be implemented, if necessary at the application level.

The radical difference between V and *Clouds* is the paradigm used by *Clouds*.

8.7. Mach

Mach is a distributed operating system under development at Carnegie Mellon [Ac86]. Mach maintains object-code compatibility with Unix. Mach extends the Unix paradigms by adding large sparse address spaces, memory mapped files, user provided backing stores, and memory sharing between tasks. Mach is implemented on a host of processors including multiprocessors.

The execution environment for a Mach activity is a task. Threads are computation units that run in a task. A single thread in a task is similar to a Unix process. Ports are communication channels, supporting messages which are typed collection of data objects. In addition, Mach supports memory objects, which are collections of data objects managed by a server.

Support for transactions are not built into Mach, but can be layered on top of Mach and has been implemented by Camelot and Avalon [HeWi87].

The approaches used by Mach and *Clouds* are fundamentally different, as with V and *Clouds*.

9. Concluding Remarks

Clouds provides an environment for research in distributed applications. By focusing on support for advanced programming paradigms, and decentralized, yet integrated, control, *Clouds* offers more than 'yet another Unix extension/look-alike'. By providing mechanisms, rather than policies, for advanced programming paradigms, *Clouds* provides systems researchers an adaptable, high-performance, 'workbench' for experimentation in areas such as distributed databases, distributed computation, and network applications. By adopting 'off the shelf' hardware, the portability and robustness of *Clouds* are enhanced. By providing a 'Unix gateway', users can make use of established tools. The gateway also relieves *Clouds* from the necessity of providing emulating services such as provided by Unix mail and text processing.

The goal of *Clouds* has been to build a general purpose distributed computing environment, suitable for a wide variety of user communities, both within and outside the computer science community. We are striving to achieve this through a simple model of a distributed environment with facilities that most users

would feel comfortable with. Also we are planning to experiment with increased usage of the system by making it available to graduate courses, and hope the feedback and the criticism we receive from a large set of users will allow us to tailor, enhance and perhaps redesign the system to fit the needs for distributed computing, and thus give rise to wider usage of distributed systems.

10. Acknowledgements

The authors would like to acknowledge Martin McKendry and Jim Allchin for starting the project and designing the first version of Clouds. Gene Spafford and Dave Pitts for the implementation, Jose Bernabeau, Yousef Khalidi and Phil Hutto for their efforts in making the kernel usable and for the design of *Ra*. Also Mustaque Ahamad, Ray Chen, Kishore Ramachandran and Henry Strickland for their participation in the project.

11. References

[Ac86] Accetta M, et. al. *Mach: A New Kernel Foundation for Unix Development*, Technical Report, Carnegie Mellon University.

[AhAm87] M. Ahamad, M. Ammar, J. Bernabeu and M. Y. Khaldi, *A Multicast Scheme for Locating Objects in a Distributed System*. Technical Report GIT-ICS-87/01, School of Information and Computer Science, Georgia Tech, January 1987.

[AhDa87] M. Ahamad and P. Dasgupta, *Parallel Execution Threads: An Approach to Atomic Actions*. Technical Report GIT-ICSD-87/16. School of Information and Computer Science, Georgia Tech.

[Alm83] G. T. Almes, *The Evolution of the Eden Invocation Mechanism*, Technical Report 83-01-03, Department of Computer Science, University of Washington, 1983.

[Al83] J. E. Allchin, *An Architecture for Reliable Decentralized Systems*, Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, (Also released as technical report GIT-ICS-83/23.) 1983.

[AlBl83] G. T. Almes, A. P. Black and E. D. Lazowska and J. D. Noe, *The Eden System: A Technical Review*, University of Washington Department of Computer Science, Technical Report 83- 10-05 October 1983.

[BeHuKh87] J. M. Bernabeau Auban, P. W. Hutto and M. Y. A. Khalidi, *The Architecture of the Ra Kernel*, Technical Report GIT-ICS-87/35 School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987.

[BeRc85] J. C. Berets, R. A. Mucci and R. E. Schantz, *Cronus: A Testbed for Developing Distributed Systems*, October 1985 IEEE Communications Society, IEEE Military Communications Conference.

[Bi85] K. P. Birman and others, *An Overview of the ISIS Project*, Distributed Processing Technical Committee Newsletter, IEEE Computer Society (7,2) October 1985 (Special issue on Reliable Distributed Systems).

[Ch88] D. Cheriton *The V Distributed System*. Communications of the ACM, March 1988.

[ChDa87] R. Chen and P. Dasgupta, *Consistency-Preserving Threads: Yeat Another Approach to Atomic Programming*. Technical Report GIT-ICS-87/43 School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987.

[Da86] P. Dasgupta, *A Probe-Based Fault Tolerant Scheme for an Object-Based Operating System*, Proceeding of the 1st ACM Conference on Object Oriented Programming Systems, Languages and Applications. Portland OR. 1986.

[GuDc86] R. F. Gurwitz, M. A. Dean and R. E. Schantz, *Programming Support in the Cronus Distributed Operating System*, May 1986, Proceedings of the Sixth International Conference on Distributed Computer Systems, IEEE Computer Society.

[GrScWc86] I. Greif, R. Seliger and W. Weihl *Atomic Data Abstractions in a Distributed Collaborative Editing System*, (Extended Abstract) Conference Record of the Thirteenth Symposium on Principles of Programming Languages, ACM SIGACT/SIGPLAN, January 1986, St. Petersburg Beach, FL.

[HeWi87] M. P. Herlihy and J. M. Wing, *Avolon: Language Support for Reliable Distributed Systems*. Proceedings of the 17th International Symposium on Fault-Tolerant Computing. July 1987.

[Jc85] E. D. Jensen et. al. *Decentralized System Control*. Technical Report RADC-TR-85-199, Carnegie Mellon University and Rome Air Development Center, April 1985.

[LeWi85] R. J. LeBlanc and C. T. Wilkes, *Systems Programming with Objects and Actions*, Proceedings of the Fifth International Conference on Distributed Computing Systems, Denver, July 1985. (Also released, in expanded form, as technical report GIT-ICS-85/03)

[LiSc83] B. Liskov and R. Scheifler, *Guardians and Actions: Linguistic Support for Robust Distributed Programs*, ACM, Transactions on Programming Languages and Systems (53) July 1983.

[Li87] B. Liskov, D. Curtis, P. Johnson and R. Scheifer. *Implementation of Argus*. Proceedings of the 11th ACM Symposium on Operating Systems Principles. November 1987.

[Mc84] M. S. McKendry, *Clouds: A Fault-Tolerant Distributed Operating System*, Distributed Processing Technical Committee Newsletter, IEEE, 1984, (Also issued as Clouds Technical Memo No:42).

[Mo81] J. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.

[NoPr85] J. D. Noe, A. B. Proudfoot and C. Pu, *Replication in Distributed Systems: The Eden Experience*, Department of Computer Science, University of Washington, Seattle, WA, September 1985 Technical Report TR-85-08-06.

[No87] Northcutt J. D. *Mechanisms for Reliable Distributed Real-Time Operating Systems - The Alpha Kernel*, Perspectives in Computing, v16. Academic Press, 1987.

[Pi86] D. V. Pitts, *Storage Management for a Reliable Decentralized Operating System*, Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986, (Also released as Technical Report GIT-ICS-86/21).

[RaKh88] U. Ramachandran and Y. A. Khalidi, *Memory Management Support for Object Invocation*. Technical Report GIT-ICS-88/03. School of Information and Computer

Science, Georgia Tech.

[ScTh86] R. E. Schantz, R. H. Thomas and G. Bono, *The Architecture of the Cronus Distributed Operating System*, May 1986, Proceedings of the Sixth International Conference on Distributed Computer Systems, IEEE Computer Society.

[Sp86] E. H. Spafford, *Kernel Structures for a Distributed Operating System*, Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986, (Also released as technical report GIT-ICS-86/16).

[WeLi83] W. Weihl and B. Liskov, *Specification and Implementation of Resilient Atomic Data Types*, Symposium on Programming Language Issues in Software Systems, June 1983.

[Wi85] C. T. Wilkes, *Preliminary Aeolus Reference Manual*, Technical Report GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985. (Last Revision: 17 March 1986)

[WiLe86] C. T. Wilkes and R. J. LeBlanc, *Rationale for the Design of Aeolus: A Systems Programming Language for an Action/Object System*, Proceedings of the IEEE Computer Society 1986 International Conference on Computer Languages. (Also available as Technical Report GIT-ICS-86/12, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986.)

[Wi87] C.T. Wilkes *Programming Methodologies for Resilience and Availability*. Ph.D.thesis, Georgia Tech, 1987, Technical Report GIT-ICS-87/32 School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA.

[Wu74] W. A. Wulf and others, *HYDRA: The Kernel of a Multiprocessor Operating System*, Communications of the ACM, (17,6) June 1974.

[WuLe81] W. A. Wulf, R. Levin and S. P. Harbison, *HYDRA/C.mmp, An Experimental Computer System*, McGraw-Hill, Inc., 1981.