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

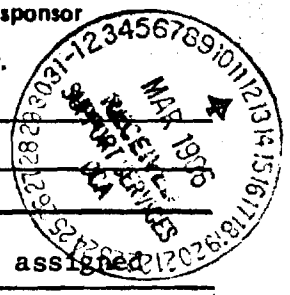
(or) Company/Industrial Proprietary: N/A

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.

SPONSOR'S I. D. NO. 02.104.000.86.005

COPIES TO:

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

12/88

GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION

NOTICE OF PROJECT CLOSEOUT

Date 12/9/88

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

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

Sponsor Air Force

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

Prime Contract No. N/A

Title Fault Tolerant Software Technology for Distributed Computing System

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

Closeout 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 ____

Includes Subproject No(s). N/A

Subproject Under Main Project No. ____

Continues Project No. ____ Continued by Project No. ____

Distribution:

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

__X__ Reports Coordinator (OCA)
__X__ GTRC
__X__ Project File
__X__ Contract Support Division (OCA)
____ Other ____

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: 18 Feb 86 - 30 Mar 86

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECNHOLOGY
ATLANTA, GEORGIA 30332

1. Progress

During this initial reporting period, organization of the project has begun to be established. Because the start of the project occured in the middle of an academic year, the level of effort will be relatively low until the summer, due to lack of availability of personnel.

Our efforts thus far have been concentrated on Task 1 of the project, Programming Techniques for Resilience and Availability. Our work on this task will be strongly related to other work involving the Principal Investigator within the Clouds Project at Georgia Tech.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

Dr. Richard LeBlanc is directing this project, as specified in the Key Personnel section of the contract.

Mr. Tom Wilkes is employed by the project as a graduate research assistant. He is a PhD student in the School of Information and Computer Science at Georgia Tech. His recent research efforts have been concerned with the design and implementation of a programming language (called Aeolus) which includes features which provide access to the action and object management features of the Clouds kernel. These language features are to provide the basis of our study of programming techniques for resilience and availability.

4. Summary of Trips and Meetings

None during the reporting period.

5. Problems or Areas of Concern

No problems or areas of concern are evident at the current time.

6. Sufficiency of Effort Towards Meeting Goals of the Contract

The current level of effort is well below that needed to meet the goals of the contract. Our level of effort will increase substantially over the next four months, thereby reaching a sufficient level.

7. Related Accomplishments

There have been no related accomplishments during this period.

G-36-645

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: April, 1986

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Our work on Task 1 of the project, Programming Techniques for Resilience and Availability, has continued. A study of related research work is in progress. Work on Task 2 has not started yet; we plan to initiate that effort in June or July.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

No new personnel have been added to the project. A graduate research assistant has been identified to work on Task 2 beginning in the summer quarter. He is Mr. Steve Ornburn.

4. Summary of Trips and Meetings

Dr. LeBlanc visited RADC to meet with relevant staff members to discuss this contract and the RADC research program in distributed systems. He presented an overview of the Clouds project and discussed the planned research and projected work schedule. The summary of RADC supported research presented by R. Metzger suggested several contacts that should be made with groups doing related research.

5. Problems or Areas of Concern

No problems are areas of concern are evident at the current time

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is well below that needed to meet the goals of the contract. Our level of effort will increase substantially over the next three to four months, thereby reaching a sufficient level.

7. Related Accomplishments

There have been no related accomplishments during this period.

8. Plans for Next Period

The study of related work under Task 1 will continue. We will also begin designing example problems to be used in our methodology studies.

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: May, 1986

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Our work on Task 1 of the project, Programming Techniques for Resilience and Availability, has continued. Our study of related research work has led us to conclude that most work with replicated objects has been concerned with algorithms to control object interactions. Little has been done to explore state replication methods. We plan to make that area an initial focus for our study. Work on Task 2 has not started yet; we plan to initiate that effort in July.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

No new personnel have been added to the project.

4. Summary of Trips and Meetings

Dr. LeBlanc attended a NASA-sponsored workshop on Embedded Distributed Computer Systems in Tampa on May 8 and 9. The program of this workshop consisted of presentation and discussion research reports by a small group of NASA-funded researchers. A variety of interesting research topics were discussed. The most valuable result of this trip was the development of some ideas about how Clouds might be made interoperable with Unix, so as to quickly provide a powerful development environment for Clouds.

Dr. LeBlanc also attended the Distributed Computing Systems conference in Boston on May 20-22. During this trip he heard two presentations on Chronus by BBN personnel and had opportunities for further discussions with them. On May 23, he gave a presentation about Clouds at Computer Corporation of America and discussed various research topics with David Reiner and Sunil Sarin.

5. Problems or Areas of Concern

No problems are areas of concern are evident at the current time

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is well below that needed to meet the goals of the contract. Our level of effort will increase substantially over the next three to four months, thereby reaching a sufficient level.

7. Related Accomplishments

There have been no related accomplishments during this period.

8. Plans for Next Period

Methods of state replication, particularly for nested objects, will be the focus of our study under Task 1. Appropriate example problems for this study will be identified.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	8.5	34
Research Scientist II	0	0
Grad. Research Asst.	87	304.5
Secretary	60	120
Clerk Typist	60	120

G-36-645

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: June, 1986

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Our work on Task 1 of the project, Programming Techniques for Resilience and Availability, has continued. 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 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), 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, we say that B is 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. 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 the computation from which the desired state results on each replica; we refer to this scheme as idemexecution. Another method is to execute the computation at one replica, and then copy the state of that replica to the other replicas; we refer to this scheme as cloning. 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 our investigation.

Our current research includes an investigation of a ''taxonomy'' of object structures on which the corresponding state-propagation methods may be safely used, as well as of how these state-propagation methods -- or the Clouds object-naming mechanism -- may be altered to safely handle more general cases.

Work on Task 2 has not started yet; we plan to initiate that effort in July.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

No new personnel have been added to the project. However, a Research Scientist has been selected to join the project in October. He is Win Strickland, who holds an M.S. from Georgia Tech. He has considerable expertise with operating systems, having worked in the ICS Laboratory while he was a student and for three years with a startup company in the Atlanta area.

4. Summary of Trips and Meetings

Dr. LeBlanc attended the SIGPLAN '86 Symposium on Compiler Construction in Palo Alto, California. There were several significant papers presented there concerning techniques for implementing attribute grammar-based programming tools. This work is relevant to ongoing efforts within the Clouds project towards the implementation of tools for using Aeolus.

5. Problems or Areas of Concern

No problems are areas of concern are evident at the current time

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is below that needed to meet the goals of the contract. Our level of effort will increase substantially next month and again when more personnel are added in October, thereby reaching a sufficient level.

7. Related Accomplishments

There have been no related accomplishments during this period.

8. Plans for Next Period

A principal focus of our work will be exploring the concept of an object filing system that can serve as a repository and access mechanism for external objects (as defined in section 1 above).

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	36.5	70.5
Research Scientist II	0	0
Grad. Research Asst.	87	391.5
Secretary	60	180
Clerk Typist	60	180

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: July, 1986

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Under Task 1 of the project, Programming Techniques for Resilience and Availability, we have continued our development of a taxonomy of object structures on which the alternative state-propagation methods may be safely used. Additionally, we are considering how these state-propagation methods -- or the Clouds object-naming mechanism -- may be altered to safely handle more general cases.

Work on Task 2, Action-Based Programming for Embedded Systems, has now started. We have identified the work of the ISIS project at Cornell as significant to our investigation and are currently studying their results.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

A new graduate research assistant, Stephen Ornburn, joined the project this month. After he is currently familiarizing himself with our ongoing work and focussing on Task 2.

4. Summary of Trips and Meetings

None.

5. Problems or Areas of Concern

No problems are areas of concern are evident at the current time

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is still below that needed to meet the goals of the contract. Our level of effort will increase substantially when more personnel are added in October, thereby reaching a sufficient level.

7. Related Accomplishments

There have been no related accomplishments during this period.

8. Plans for Next Period

We will continue the work described above for Task 1 and we will be exploring the concept of an object filing system that can serve as a repository and access mechanism for external objects. For Task 2, we will be studying the ISIS work.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	59	129.5
Research Scientist II	0	0
Grad. Research Asst.	174	565.5
Secretary	10	190
Clerk Typist	10	190

G-36-645

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: August, 1986

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Under Task 1 of the project, Programming Techniques for Resilience and Availability, we have been studying the work of Herlihy, presented in his PhD thesis, "Replication Methods for Abstract Data Types." We have discovered a correspondence between his quorum intersection graphs and the lock compatibility tables of Aeolus. This discovery is valuable because it should allow us to apply techniques developed by Herlihy to our problem of automatically generating replicated objects from a single object version and a replication specification.

Work on Task 2, Action-Based Programming for Embedded Systems, has continued this month with an examination of the consequences of irreversible nested actions in action-based programs. Modelling dependency of other actions on irreversible actions will be a major concern. We intend to look at models from database dependency theory for application to this problem. We have also identified techniques for forward error recover (as opposed to the rollbacks normally used in action-based programs) as an important topic for study.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

No changes.

4. Summary of Trips and Meetings

None.

5. Problems or Areas of Concern

No problems or areas of concern are evident at the current time.

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is still below that needed to meet the goals of the contract. Our level of effort will increase substantially when more personnel are added in October, thereby reaching a sufficient level.

7. Related Accomplishments

There have been no related accomplishments during this period.

8. Plans for Next Period

We will continue the work described above for Task 1 and we will be exploring the concept of an object filing system that can serve as a repository and access mechanism for external objects. For Task 2, we will be studying work on forward error recovery.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	74.5	204
Research Scientist II	0	0
Grad. Research Asst.	174	739.5
Secretary	10	200
Clerk Typist	10	200

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: September, 1986

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Under Task 1 of the project, Programming Techniques for Resilience and Availability, we are currently investigating the design of the object filing system (OFS) for Clouds. The replication scheme which we are currently considering in support of availability will require heavy interaction between the manager for a replicated object, the job scheduler, and the OFS. The OFS should:

- be resilient and highly available (through replication);
- provide a mapping from object names (strings) to Clouds object capabilities;
- impose some familiar structure (e.g., a Unix-like hierarchical structure) on the flat, global system name space provided by the Clouds object manager;
- provide efficient forms for the most common types of I/O (such as text I/O) without the necessity of the context switches which would be required if such I/O were modelled with Clouds objects.

In the OFS, an object name may represent a group of objects (the set of replicas of a replicated object), rather than a single instance. We intend that this mechanism should be, in general, transparent to the user (although specialpurpose applications, such as DBMSs, may require that, in addition, finer control of replication be available than that provided by a general mechanism).

We are currently considering two different capability-based naming schemes which may be used by the OFS in support of state cloning, as described in a previous report. The first scheme requires minimal changes to the kernel, but relies on facets of the Clouds object look-up 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. (We do not consider this constraint 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. This constraint may be enforced by the OFS.) 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.
Although we will thus have 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 mentioned in the June report (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. We call this scheme vertical replication, since it maintains the grouping of internal objects with their parent object.

The other naming scheme makes fewer assumptions about the object look-up 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 from, for instance, 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, we refer to the scheme as horizontal replication.

Our initial design of the OFS is concerned with an unreplicated version; when completed, the design will be extended to a replicated version by use of the "distributed lock" mechanism and an analysis of the desired replication properties of the OFS.

Work on Task 2, Action-Based Programming for Embedded Systems, has continued this month with a study of recovery mechanisms. The goal of a recovery mechanism in a fault tolerant system is to return an erroneous computation to a state from which computation can continue. (We term such a state "consistent"). In principle, a programmer can achieve fault tolerance by explicitly saving state information and providing enough alternative paths for execution. The objective of research in fault tolerant computing is to design operating system and run time system support which, in conjunction with appropriate programming language constructs, will make recovery "invisible" to the main computation. In this way, we hope to realize many of the same advantages of "information hiding" frequently claimed for modular programming and the use of abstract data types.

In software designed using nested atomic actions (and with at most trivial interactions with its environment) recovery can be achieved simply by restoring computation to an earlier state and applying a different action. If a software system has interacted with its environment in irreversible ways, then returning the computation to an earlier state is not appropriate. The recovery mechanism must restart the computation on a state which is both consistent and in which the irreversible actions are shown as having occured. We term a recovery mechanism with these capabilites as a "forward" recovery mechanism. Initially, we are modeling recovery in the face of irreversible actions as a four step process:

1. identify the irreversible actions nested within the action to be aborted;
2. save the results of the nested, irreversible actions;
3. rollback the nested, reversible actions;
4. apply forward recovery to the nested, irreversible actions so as to establish a consistent state.

Step two raises some issues with respect to operating system and run-time system support. In particular the recoverable area techniques provided in Clouds will not be adequate. We will need to be able to selectively recover various combinations of actions.

Recovery in the face of irreversible actions presupposes that the irreversible actions can themselves be identified. Among the irreversible actions, we will of course have the action which interacted with the environment in an irreversible way. Labeling an action "irreversible" may, however, require that other actions be labeled "irreversible" as well. For example, consider three actions A, B, and C. A and B are nested within C. B is irreversible, and A provides data to B. In the event that C aborts, it will not be possible to undo the effects of A. Since B is irreversible and its effects will survive step three. Even if A itself is rolled back, that A had occured may still be evidenced by the results of B. Undoing A and not B produces an inconsistent state in which A has both occured and not occured. We choose to solve this problem by considering actions which have supplied data to irreversible actions as themselves irreversible. Thus, when action C is aborted, the forward recovery mechanism invoked in step four will be charged with handling both A and C.

A programmer may wish to define other actions as irreversible as well. Suppose B and D are nested within C. Suppose D uses the results of B to calculate some necessary consequences of B. The programmer wants any state which includes B to also include D. If B is labeled irreversible then D should be as well. We have identified two ways in which actions may makes other actions irreversible. We write A -¶ B to indicate that if action A is irreversible then action B is as well. Determining which of a set of nested actions are irreversible is an inference problem similar to the problem in database design theory of infering functional dependencies. Depending on the combination of actions which have been labeled irreversible, different strategies for forward recovery may be indicated.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

No changes.

4. Summary of Trips and Meetings

None.

5. Problems or Areas of Concern

No problems or areas of concern are evident at the current time.

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is still below that needed to meet the goals of the contract. Our level of effort will increase substantially when an additional 1/2-time Graduate Research Assistant and a 3/4-time Research Scientist begin working on the project in October, thereby reaching a sufficient level. A staffing projection for the rest of the project is attached to this report.

7. Related Accomplishments

Considerable progress has been made on debugging the Clouds kernel during the last three months. We expect to have all of the currently implemented functionality fully operational in October. This will include creation of object instances from templates created by the Aeolus compiler, invocation of object operations on remote machines using the standard Clouds location-transparent invocation protocol, and network communication features allowing interaction with Unix machines. The implementation of the action manager is currently in progress, based on a detailed design completed earlier in the year.

8. Plans for Next Period

We will continue the work described above for Task 1, particularly concentrating on a distributed version of the object filing system. For Task 2, the next step in our research will be to develop specification mechanisms which support our strategy for identifying irreversible actions and for selecting appropriate forward recovery strategies. We will also look more closely at specific forward recovery strategies and develop some examples illustrating our approach.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	74.5	278.5
Research Scientist II	0	0
Grad. Research Asst.	174	913.5
Secretary	10	210
Clerk Typist	10	210

Projected Level of Effort by Months

Each monthly entry gives planned man-hour expenditure for the month and total expended though the end of that month.

	Expended as of 9/30/86	10/86	11/86	12/86	1/87	2/87	3/87
Associate Professor		38.5	38.5	38.5	38.5	38.5	38.5
	278.5	317	355.5	394	432.5	471	509.5
Research Scientist		130.5	130.5	130.5	130.5	130.5	130.5
	0	130.5	261	391.5	522	652.5	783
Grad Research Asst.		261	261	261	261	348	348
	914	1175	1436	1697	1958	2306	2654
Secretary/Clerical		87	87	87	87	87	87
	420	507	594	681	768	855	942

	Expended as of 3/31/87	4/87	5/87	6/87	7/87	8/87	9/87
Associate Professor		38.5	38.5	38.5	140	140	140
	509.5	548	586.5	625	765	905	1045
Research Scientist		130.5	130.5	130.5	130.5	130.5	130.5
	783	913.5	1044	1174.5	1305	1435.5	1566
Grad Research Asst.		348	348	348	348	348	348
	2654	3002	3350	3698	4046	4394	4742
Secretary/Clerical		87	87	87	87	87	87
	942	1029	1116	1203	1290	1377	1464

	Expended as of 9/30/87	10/87	11/87	12/87	1/88	2/88	Contract Total
Associate Professor		35	35	35	35	35	1218
	1045	1080	1115	1150	1185	1220	
Research Scientist		130.5	130.5	130.5	130.5	0	2088
	1566	1696.5	1827	1957.5	2088	2088	
Grad Research Asst.		348	348	348	348	174	6264
	4742	5090	5438	5786	6134	6308	
Secretary/Clerical		87	87	87	87	87	1914
	1464	1551	1638	1725	1812	1899	

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: October, 1986

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Task 1 (Programming Techniques for Resilience and Availability):
In our last report, we described our study of the work of Herlihy concerning the efficient support of replication for abstract data types. We also described a connection between the quorum-intersection graphs used in Herlihy's work and the specification of lock mode compatibilities supported by the Aeolus/Clouds lock construct. The graph derived (using Herlihy's conventions) from the mode compatibility specification for a lock which is used to enforce synchronization among an object's operations is approximately the complement of the quorum-intersection graph for that object's operations. Thus, we should be able to use Herlihy's methods in conjunction with our lock specifications, which will be useful in light of Herlihy's optimality results for his scheme.

To test this conjecture as well as the possibility of deriving replicated objects from single-copy versions, we have been occupied during the past month with the specification and implementation of a single-copy version of a prototype Object Filing System (OFS) for Clouds. (The functionality of the OFS has also been described in a previous report; a much-simplified version of an OFS supporting a flat name space has already been implemented in the Clouds system, while the current OFS effort will support a hierarchical name space.) This effort involves the specification of the synchronization behavior of the single-copy OFS object via a lock compatibility matrix. We will compare the graph derived from this specification with the quorum-intersection graph appropriate to the same set of operations. Also, when the specification of the single-copy OFS object is complete, we will test our idea of extending the single-copy version to a replicated version by allowing the locks specified for the single-copy version to act as "distributed locks" (where information about locks granted or released by a replica on one node is communicated to the other nodes where replicas exist by the Clouds object manager, as described in the previous report). A "distributed lock" may be viewed as a manager for gathering a quorum for a given operation; the synchronization behavior thus achieved should reflect that specified for the single-copy version, with no additional effort on the programmer's part.

Another interesting question which will be investigated using the OFS example is the relation between Herlihy's quorum intersection graphs and Aeolus/Clouds lock compatibility matrices when multiple locks are used for specifying an object's synchronization behavior. We have found that in certain cases it is convenient to use more than one lock to specify synchronization among an object's operations; the different locks typically apply at differing levels of granularity as well as having compatibility matrices with disjoint meanings. For example, we have designed a symbol table object which uses two locks for synchronization purposes: one lock at the level of the individual buckets in the symbol hash table, with compatibilities expressing a multiple reader / single writer protocol; and another lock at the level of the entire symbol table, allowing multiple readers or multiple writers, but not readers concurrently with writers. The first lock is used with the typical operations such as insert, delete, and find, where there is no interaction between concurrent operations on different buckets; the second lock is used with an "exact-list" operation, where a "snapshot" of the exact state of the symbol table at a particular instant is desired, and thus all operations which modify the state of any

portion of the symbol table must be locked out. Our locks thus have an advantage of power of expression over Herlihy's quorum intersection graphs, which do not allow the expression of granularity lower than that of an entire abstract data type and its operations. Thus, we will be considering how Herlihy's results may be extended to the case of multiple levels of granularity for synchronization.

Task 2 (Action-Based Programming for Embedded Systems):
When a computation interacts with its external environment, the computation will perform operations not only on data objects belonging to the computation but also on objects belonging to the system in which the computation is embedded. These operations may cause the external object to change state in an irreversible way. This is most easily understood when the external object is real. Even when the containing system is itself computational, however, it may be convenient to regard operations performed by the embedded system on objects in the containing system as irreversible from the perspective of the embedded system.

Our proposed strategy for achieving fault-tolerance in the presence of irreversible operation involves aborting the action in which the fault occurred and restoring the object on which the action was begin performed to a consistent state. As the action is aborting we will save those portions of the object's state associated with the irreversible operation and then recover the object's previous state. We will then modify the recovered state to reflect the nested irreversible actions. Control returns to the point just prior to the invocation of the failed action.

We must be careful here: we want actions to be atomic, but when we modify the recovered state to reflect the irreversible operations performed by the failed action we appear to lose atomicity. From the perspective of the main computation, however, there is another interpretation -- the object appears to have undergone a spontaneous state change.

If we allow these "spontaneous" state changes, we must do more as well. The correct execution of a program often depends on the state of the computation satisfying certain invariants at particular junctures. Programmers often use their knowledge of the invariants when writing a component of a program and thereby produce a simpler piece of code. The risk is that the "spontaneous" state changes may alter the state in a way that violates an invariant which a programmer had assumed to hold.

We propose to solve this problem by extending the action/object model of computation used in Clouds by introducing the notion of "triggered actions." A triggered action may be executed when entering or exiting a standard action. The triggered action will only be executed when its trigger condition is satisfied. Among the trigger conditions which may be specified is the "spontaneous" state change of an object. As we have seen the spontaneous state change may be a side effect of recovery in the presence of irreversible operations. Spontaneous state changes may also occur in dynamic environments which are not entirely under the control of the computation. The triggered action construct may be of use when constructing software which must interact with an environment in which genuine spontaneous state changes occur.

Our intention is to let the triggered action appear to the main computa-

tion as the putative source of the updates to object states. The programmer using triggered actions can think of them as triggered by an events external to the software and causing spontaneous state changes to the data objects. The triggered action will be charged with the responsibility for carrying out any bookkeeping required in response to the spontaneous state change. The spontaneous state change in a sense provides an alternate path through a section of code and the triggered action can be used to ensure that the invariants are preserved. We also will allow the programmer to indicate that certain operations or sequences of operations should be regarded as irreversible. There should be programmer control over the activation and deactivation of triggered actions. There should also be a means for establish the order in which active triggered actions are executed when the triggers of several are satisfied simultaneously.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

Win Strickland has joined the project staff as a Research Scientist and Scott Vorthmann has begun working as a 1/2-time Graduate Research Assistant.

4. Summary of Trips and Meetings

Dr. RIchard LeBlanc attended the RADC Technology Exchange during this month. This meeting provided valuable contacts with several other groups doing related work, particularly the Cronus group at BBN and the Honeywell fault-tolerance group.

Dr. LeBlanc and Tom Wilkes also attended the IEEE International Conference on Computer Languages, where Wilkes presented a paper on the design of Aeolus and LeBlanc chaired a panel session on programming models for distributed computing. The principal benefits of this trip were wider exposure for the Clouds project and our discussions with researchers working in the area of specification-based programming.

5. Problems or Areas of Concern

No problems or areas of concern are evident at the current time.

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is now sufficient to meet the goals of the contract, according to the plan included with last month's report.

7. Related Accomplishments

The Clouds kernel is now operational, with the exception of the action manager. A small amount of work remains to be done in the area of run-time support for Aeolus. An interface to Unix has been constructed, allowing us to make effective use of the multiple window displays available on Sun workstations for debugging.

8. Plans for Next Period

We will continue the work described above for Task 1, expecting to complete the implementation effort for the single-copy OFS object within a few weeks. We may thus be able to give preliminary results of our analysis of the behavior of this version as extended to a replicated version in our next report. For Task 2, our next step is to develop the language mechanims for specifying that operations are irreversible and for defining and controlling triggered actions. We will also consider in more detail the ways in which programmers exploit their knowledge of invariants when constructing sections of a program; from this study we will be able to suggest guidelines for using triggered actions.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	38.5	317
Research Scientist II	130.5	130.5
Grad. Research Asst.	261	1175
Secretary	47	257
Clerk Typist	40	250

G-36-645

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: November, 1986

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Task 1 (Programming Techniques for Resilience and Availability):
In our last report, we described the implementation in Aeolus of a single-copy version of a prototype Object Filing System (OFS) for Clouds. The idea is to use this unreplicated version to design the lock compatibility matrix necessary for its synchronization, and then to study the relationship of the compatibility matrix with the quorum intersection graph derived when the single-copy version is extended to a replicated version.
The prototype OFS design involves a hierarchical nesting of OFS objects, each of which maintains knowledge of its immediate ancestor in the hierarchy. (Here, by "nesting" we mean "logical nesting," that is, an object contains a capability to its child, as opposed to "physical nesting," where an object physically contains its child.) The children of an OFS are stored in and accessed through a symbol table object nested in the OFS. The design of the symbol table object has been described in one of our publications ("Systems Programming with Object and Actions," R.J. LeBlanc and C.T. Wilkes, Proceedings of the 5th International Conference on Distributed Computing Systems); the locking structure of the symbol table object was described briefly in our last report.

The interpretation of user commands to the OFS is handled by a rudimentary "shell" process, which accepts Unix-like pathnames and translates them to operations on an OFS or invocations of Aeolus processes. The shell process maintains knowledge of the root of the OFS hierarchy, as well as the "current" OFS (corresponding to the "current working directory" in Unix), and (for efficiency purposes) the ancestor of the current OFS.

Since our last report, we have done additional work on the prototype OFS design, though it is not yet complete. During this month, our efforts have also included work on the implementation and debugging of the Aeolus run-time support necessary for the support of Aeolus/Clouds objects and their interaction with the Clouds kernel; this effort, along with some additional compiler and language library implementation effort, should allow us to run the prototype OFS under Clouds shortly after the prototype is complete. Although the OFS design and implementation are nearly finished, we prefer to complete the implementation of all OFS operations before adding synchronization, since the design of the lock compatibility matrices often depends on subtle interactions among the object operations. However, our current feeling is that the synchronization mechanism already in place in the nested symbol table objects (as described in our previous report) may suffice for the synchronization of the OFS objects as well. This would simplify the analysis of the compatibility matrix/quorum intersection graph relationships, since the symbol table object has only five operations, compared to ten for the OFS object, and thus fewer interactions among the object's operations, yielding simpler compatibility matrices. Since the symbol table object synchronization involves two locks at differing levels of granularity, we will be able to investigate how such locks relate to quorum intersection graphs when the latter are extended to multiple levels of granularity.

Task 2 (Action-Based Programming for Embedded Systems):
We have proposed a construct called a "triggered action" which will -- following the rollback of an action which failed after performing an irreversible operation -- restore the computation to a consistent state which

recognizes that the irreversible operation occured. We have refined our notion of a "consistent state" and have explored some of the issues in using triggered actions to establish a consistent state. Actions, our computational units, can be coupled together in various ways so that the results of one action affect the results of the another. The constructs for coupling actions together are provided by the underlying programming language; in general though, two actions are coupled to permit the flow of data and control information across their boundary. For our purposes we will regard the flow as occuring in one direction.

Researchers have described several forms of coupling including content, common, control, stamp and data coupling. For our purposes it is sufficient to observe that we can develop an invariant characterizing the state of a computation at the boundary between two actions. More precisely, this invariant is a formal characterization of the state of the computation just after control passes the end of the first action. We may also use invariants to characterize the entry conditions of the second action. In a correctly constructed program, if the invariant at the boundary between two actions is satisfied, then the entry conditions of the second action are of necessity also satisfied. When implementing software, programmers often deliberately exploit their knowledge of program invariants in order to speed up or simplify their programs. The assumptions a programmer makes about the state of a computation at a particular boundary constitute the constraints on the state of the computation and it is with respect to these constraints that we judge the consistency of a state of a computation. When recovering from a failed action, we must ensure that the state of the computation is consistent with the constraints associated with the point in the program at which execution is resumed. When recovering from a failed action which executed an irreversible operation, we must update the state of the computation to indicate that the irreversible operation occured. It is the responsibility of the triggered action to ensure that these updates are sufficient to satisfy the constraints associated with the point in the program at which execution will be resumed.

Programmers often construct the first in a pair of actions so as to simplify the invariant at the boundary between the two actions, thus making it easier to use. The tight coupling of a pair of actions is often labeled "tricky programming." The connotation is that while the tricks may make the program simpler to write or more efficient, the tricks also make the program harder to understand, debug and modify. The negative aspects of tightly coupled actions arise in part because there is no standard way of documenting them. There appears to be a need to consider a notation for documenting the constraints which must be preserved at the boundary between two actions. If the constraints were documented, then when programmers add new execution paths to the first action they would have a better guide as to the constraints which must be satsified when exiting the action. In particular, triggered actions can be viewed as providing alternative paths of execution. By documenting the constraints, programmers would have a better guide for constructing the triggered actions. For an illustration of the problem of preserving consistency and of the use of triggered actions, consider the following example. Suppose a program consists of two components Y and Z. These components will be executed sequentially (i.e., Y;Z). Among other activities Y searches a complex data structure and sets a flag to indicate whether the target was

found. Z also needs to know whether the target is in the data structure. One solution would be to search the data structure a second time. A second solution would be to exploit our knowledge about how Y operates to decide whether the target is in the data structure. We might observe that

a. Y never resets the flag which indicates whether the target was found
b. Y only takes the target out of the data structure if property P is true
c. The variables used in evaluating property P are not subsequently changed.

With this information, Z could be constructed to decide whether the target is in the data structure by checking the flag and testing whether property P still holds. Suppose further that deleting the target from the data structure was an irreversible operation, while the assignments of values to the flag and to the variables referenced when property P is evaluated are reversible. Finally, suppose program component Y is a recoverable action.

Recovery from a failure of action Y must be done so as to ensure Z will execute correctly. Our strategy is to construct a triggered action which makes it appear as though the target was deleted spontaneously from the data structure. The triggered action will ensure the flag and the variables used in evaluating P are consistent with the state of the data structure. Only then will execution resume. Simpler couplings between components Y and Z are possible. Y could set a flag to indicate the results of evaluating property P and then Z could infer the results of evaluating P by examining the flag. The triggered action would have to be redesigned accordingly. The example illustrates some of the difficulties which can be encountered when attempting to recover actions which involve irreversible operations. It also illustrates how our approach allows a programmer to deal with these difficulties in a way that is consistent with our action-based computational model.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

No changes.

4. Summary of Trips and Meetings

During November, we hosted a site visit by staff of the National Science Foundation and two outside reviewers as part of the evaluation of a proposal by Georgia Tech to the NSF Coordinated Experimental Research Program. (One of the reviewers was Rick Schantz of BBN.) The main thrust of this proposal is a plan to make the Clouds testbed available to other experimenters within the School of Information and Computer Science. If the proposal is funded, major resources will be made available to us to further develop the Clouds system.

5. Problems or Areas of Concern

No problems or areas of concern are evident at the current time.

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is now sufficient to meet the goals of the contract.

7. Related Accomplishments

There have been no related accomplishments during this period.

8. Plans for Next Period

We will continue the work described above for Task 1, planning to complete the implementation effort for the single-copy OFS object within a few weeks. We will then be able to give preliminary results of our analysis of the behavior of this prototype as extended to include replication. For Task 2, our next step is to consider some of the issues associated with implementing our approach to acheiving fault-tolerance in the presence of irreversible operations. These issues include designing appropriate programming language syntax and the developing a notation for specifying constraints at the boundary between two actions. We will also consider in more detail some of the pragmatic issues associated with our use of triggered actions, such as failure of a triggered action, recovery in the face of multiple irreversible operations.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	38.5	355.5
Research Scientist II	130.5	261
Grad. Research Asst.	261	1436
Secretary	40	297
Clerk Typist	47	297

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: December, 1986

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Task 1 (Programming Techniques for Resilience and Availability):

In previous reports, we have described the implementation in Aeolus of a prototype Object Filing System (OFS) for Clouds, as well as the implementation of a simple shell process to exercise the OFS. The prototype OFS design involves a hierarchical (logical) nesting of OFS objects. Thechildren of an OFS are stored in and accessed through a symbol table object nested in the OFS; since a child may be another OFS as well as any other type of Clouds object, we achieve a hierarchical filing system similar in user interface to the UNIX file system.

Since our last report, we have completed testing of the run-time support needed by Aeolus to handle creation and invocation of Clouds objects. This support includes a very primitive version of the OFS, which allows Aeolus code to obtain capabilities to previously-created objects from the Clouds kernel nameserver using a flat name space. We are now integrating additional support required by the full prototype OFS implementation (primarily concerned with character-string handling) into the Aeolus runtime system. When this support is available, we will be able to test the full OFS implementation under Clouds. Concurrently, we are performing the replication analysis of the OFS as described in previous reports.

The implementation of hierarchical directory structure using nested OFS objects requires at least one invocation of a different Clouds object for each nested directory in a pathname. Under the prototype implementation of the Clouds kernel, a Clouds object invocation involves mapping that object's virtual address space into user space. We have little experience yet with the effect of this implementation of object invocation on performance. In order to study its impact, we are also implementing an alternative design of the OFS which, rather than using explicit nesting of OFS objects to obtain a hierarchical directory structure, maintains a tree structure in a single Clouds object; a non-directory object at a node is represented by a capability (as in the nested implementation), while a nested directory is represented by a pointer to another tree node. Thus, in this implementation we avoid invocations of Clouds objects during traversal of the directory structure. However, this implementation is less straightforward and transparent than the nested implementation; we lose the nice abstraction properties yielded by the separation of function into separate Clouds objects (OFS and symbol table). We could use the non-Clouds object constructs provided by Aeolus to achieve a more structured design in the non-recoverable version of the non-nested implementation; however, such a design would not be practical when we add recoverability, since access to the recoverability features of Clouds is not available from the non-Clouds objects of Aeolus, even when these objects are nested within a recoverable Clouds object. This suggests a possible deficiency in the design of Aeolus non-Clouds objects, which we intend to study further. In any case, we feel that comparison of the two OFS designs will be a valuable study in the methodology of programming under Clouds.

Task 2 (Action-Based Programming for Embedded Systems):

We have previously described the notion of triggered action. A triggered action is activated if an action fails after it has performed an irreversible operation. When activated, a triggered action is supposed to restore the computation to a consistent state, thereby allowing subsequent actions to execute correctly.

We are considering ways in which the idea of triggered actions can be integrated into action-based languages such as Aeolus. In particular, we are considering how to integrate the triggered action idea into Aeolus' event handler construct. To simplify the description, we assume Aeolus has been extended to support exception handlers. Event handlers are provided for five types of event: beginning of action, top level precommit, nested precommit, commit and abort. While default handlers are provided for each of these events, the programmer has the option, for any action, of overriding the default handler and writing his own. In particular, it is possible to override the default handler for an abort event with a handler which is sensitive to the semantics of the event. Triggered actions can be regarded as a generalization of the event handler construct. Specifically, they generalize on the event handlers for the abort event. To recognize this, what we have been calling triggered actions will now be refered to as event handlers for aborts in the presence of irreversible operations. Where context makes clear which event handlers we are refering to, we will simply refer to "the event handlers."

We will descibe some of the most important considerations which will influence our choices as to syntax and semantics. Our work is being guided by our desire to provide abstractions which programmers will find useful when conceptualizing a programming problem involving irreversible operations.

Actions cannot be rolled back once they have performed an irreversible operation. Instead, failure must be accompanied by some form of forward recovery. If the code for an action includes an irreversible operation, the action is said to be potentially irreversible. If a potentially irreversible action is nested within a second action, the second action is also regarded as potentially irreversible. A potentially irreversible action becomes irreversible as soon as it performs an irreversible operation or action. All potentially irreversible operations must be accompanied by some means for continuing the action in the event of failure. Our requirement, simply stated, is that an action cannot be rolled back once it has performed an irreversible operation; it must run to completion.

We believe a programmer would find it useful if he could distinguish between actions which completed normally from those which completed abnormally. An action completing abnormally would raise an exception visible in the calling environment. Handlers for aborts in the presence of irreversible operations would cancel the abort and carry the action to completion. It would also determine whether the action should be regarded as terminating normally or abnormally. If the handler opts for abnormal termination, it will raise the appropriate exception.

Normal and abnormal terminations provide the calling environment with different guarantees. The distinction between the two types of termina-

tion is one drawn at the programmer's discretion. From a programmer's point of view the distinction should be related to whether the invariants he expects to be satisfied by the results of the action are in fact satisfied. A normal termination should guarantee that the expected invariants hold. As we have mentioned, a handler for aborts in the presence of irreversible actions may result in a normal termination if it was able to find an appropriate continuation for the failed computation.

An abnormal termination indicates possible inconsistencies in the results and causes control to pass to the appropriate exception handler. In order for the exception handler to perform any useful work, however, it must have some information about the state of the failed action. In this case, it is the role of the event handler for the abort to record that information in a location known to the calling environment's exception handler. At the very minimum, the exception handler must know which irreversible events occured before the action terminated abnormally.

The problems of propogating this minimum information into the calling environment is closely related to the problem of activating the appropriate event handler when an action fails after it has performed an irreversible operation.

The appropriate strategy for carrying a failed action on to completion depends on the point at which it failed. If an action contains several potentially irreversible actions or irreversible operations, we must know which in fact occured before the action failed. We believe it will be sufficient to associate a flag with each occurance of a potentially irreversible action or irreversible operation. The flag is clear if its associated action or operation has not begun execution. We want to be guaranteed the flag will be set when the action or operation is completed. If failure occurs in the event handler or no event handler is provided, the action should be regarded as terminating abnormally and control should be handed over to an exception handler in the calling environment. If an exception handler for an abnormally terminated, irreversible event itself fails, then control should should pass to the appropriate event handler on the same level as the exception handler.

Some of the conditions discussed above present an unfortunate complication: it may become necessary for an event handler to deal with a partially completed, irreversible action, e.g., an irreversible action fails as does its event handler and the exception handler in the calling environment. We believe we can handle this by inspecting, perhaps recursively, the flags for the irreversible operations and actions withing the hierarcy of nested actions -- this, however, requires further investigation. The flags used to record the occurence, or non-occurence, or irreversible actions and operations may also be used to propogate that information into the calling environment. By this means, we can provide the minimum amount of information required by an exception handler dealing with an abnormally terminated action.

Since we cannot guarantee that event and exception handlers will never fail, we believe that these flags must be able to propogate up through a hierarcy of nested actions. We have not yet analized the problems this strategy may present when designing appropriate runtime support.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

No changes.

4. Summary of Trips and Meetings

None.

5. Problems or Areas of Concern

No problems or areas of concern are evident at the current time.

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is now sufficient to meet the goals of the contract.

7. Related Accomplishments

There have been no related accomplishments during this period.

8. Plans for Next Period

We will continue the work described above for Task 1, planning to test the single-copy OFS object during this month. We will also work further on our analysis of the behavior of this prototype as extended to include replication. For Task 2, we will continue to develop and evaluate the semantics of our proposed event handler construct. We also must design appropriate syntax. We plan to consider the problem of designing a similar construct which does not rely on the use of exception handlers. Finally, we will develop some examples illustrating the use of our construct.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	38.5	394
Research Scientist II	130.5	391.5
Grad. Research Asst.	261	1697
Secretary	47	344
Clerk Typist	40	341

G-36-645

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: January, 1987

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

--

1. Progress

Work on both tasks is currently focused on the guidebooks that are the primary deliverables of this project.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

No changes.

4. Summary of Trips and Meetings

None.

5. Problems or Areas of Concern

No problems or areas of concern are evident at the current time.

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is sufficient to meet the goals of the contract.

7. Related Accomplishments

None.

8. Plans for Next Period

The remaining contractual effort will be applied to finishing the guidebooks.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	35	1180.5
Research Scientist II	130.5	2083
Grad. Research Asst.	174	5886
Secretary/Clerical	87	1847

G-36-645

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: January, 1987

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Task 1 (Programming Techniques for Resilience and Availability):

In previous reports, we have described work on the specification and implementation in Aeolus of a prototype Object Filing System (OFS) for Clouds. Both a nonrecoverable and a recoverable (thus resilient) version of the OFS have been completed. In the process of doing these implementations, we have discovered and characterized a variety of different internal structures possible for each of these versions.

Integration of the Aeolus runtime support with the Clouds kernel has proven to be more difficult than originally anticipated. Thus we have not been able to test our OFS objects on the Clouds system. It is anticipated that this integration will be completed in the next month.

Task 2 (Action-Based Programming for Embedded Systems):

We have continued the development and evaluation of the semantics of our proposed event handler construct for dealing with irreversible operations within atomic actions and have begun the development of some some examples illustrating the use of our construct. We believe their is a possibility that the techniques we are developing will be useful in more general circumstances than dealing with irreversible operations.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

Chu-Chung Lin has joined the project staff as a 1/2-time research assistant, as previously discussed.

4. Summary of Trips and Meetings

None.

5. Problems or Areas of Concern

No problems or areas of concern are evident at the current time.

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is sufficient to meet the goals of the contract.

7. Related Accomplishments

There have been no related accomplishments during this period.

8. Plans for Next Period

Our efforts for the month of February will be focused on documenting our work to date in an interim technical report.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	38.5	432.5
Research Scientist II	130.5	522
Grad. Research Asst.	348	2045
Secretary	40	388
Clerk Typist	47	388

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: February, 1987

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Our efforts in both tasks were concentrated on writing our interim technical report. This effort had the greatest inpact on task 2, where it spurred us to complete the formulation of several examples that illustrate the concepts we have been developing.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

No changes.

4. Summary of Trips and Meetings

Dr. Richard LeBlanc visited the University of Utah as an invited speaker. His presentation was on the Clouds Project and our programming methodology work.

5. Problems or Areas of Concern

No problems or areas of concern are evident at the current time.

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is sufficient to meet the goals of the contract.

7. Related Accomplishments

The Aeolus run-time system has now been integrated with the Clouds kernel, so we will now be able to begin testing our objects on Clouds rather than Unix.

8. Plans for Next Period

For Task 1, we plan to complete testing on the Clouds system of the single-copy OFS object during this month. We will also work further on the extension of this prototype to include replication. For Task 2, we will continue to develop and evaluate the semantics of our proposed event handler construct. An immediate need is to design appropriate language syntax so that we can express our examples more precisely to facilitate further study.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	38.5	471
Research Scientist II	130.5	652.5
Grad. Research Asst.	348	2393
Secretary	47	435
Clerk Typist	40	428

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: March, 1987

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Task 1 (Programming Techniques for Resilience and Availability):

In our interim report and our last monthly report, we have summarized the non-replicated Object Filing System design and our plans for generating a replicated version from this single-copy version. During the past month, we have been developing the theoretical and technical framework to make this possible.

In further investigation of Herlihy's work on replication of abstract data types, we have found confirmation of our conjecture that there is a close relationship between Herlihy's quorum intersection graphs and our lock compatibility matrices. In fact, in two methods he has developed for integrating concurrency control and recovery for abstract data types, called Consensus Locking and Consensus Scheduling, The differences between these models need not concern us at present; therefore, we will refer to their common basis using the term ''Consensus Locking.''

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.

The model of objects and actions which Herlihy adopts differs from ours in several significant details. We have developed a model of concurrency control and replication management (which we have called Distributed Locking) which we propose to implement with a modified version of lock management in the Clouds kernel. Despite these differences in models of replication, we believe it will be possible to adapt Herlihy's theoretical basis to our model.

Another significant difference in Herlihy's work and our own 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 an 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. The ability to specify a domain for a lock may permit increased concurrency over locking on the object level; however (although this issue requires more scrutiny), we feel that this should not affect the availability characteristics of the lock's modes.

As mentioned above, Herlihy's Consensus Locking model integrates concurrency control and replication management for abstract data types. The Aeolus/Clouds Distributed Locking model offers a similar integration; concurrency properties are given by the specification of lock domains and compatibility matrices.

We now describe how availability properties are specified, how individual replicas are managed via the naming scheme, and how updates are propagated among the replicas. We propose that the availability properties of a replicated object be specified in a separate compiland for that object type, which we call the replication specification part (or ''rep'' part, for short). The properties specified in a ''rep'' part 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. If no lock types are declared by such an object, the corresponding ''rep'' part is explicitly null.

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.) 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 passed to the Clouds lock manager (in a manner yet to be determined), and is used by it to guide the selection of sets of replicas for Distributed locks.

The naming of replicated objects in Clouds has been discussed in our interim report. Two schemes were described: vertical replication, which uses the current capability scheme, relying on the current invocation mechanism (search locally first, then broadcast the search globally) and thus requiring that all objects internal to a replicated object reside on the same node as that object (to which we will refer as the coresidence requirement); and horizontal replication, which requires the addition of several bits to the current Clouds user capability to allow the kernel to address a single replica, but which does not require all object internal to the replicated object to reside on the same node.

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 may not generally be viable, since the coresidence requirement is 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, we must abandon vertical replication as lacking sufficient generality in its applicability. Fortunately, the horizontal replication scheme does not share this drawback.

Task 2 (Action-Based Programming for Embedded Systems):

In developing programming examples which illustrate the various fault tolerant techniques available to the programmer employing an action based design in the construction of an embedded system, we have found it necessary to design an appropriate pseudo-code. Since the Aeolus programming language provides low level support for action based programming, the pseudo-code is needed to achieve the desired level of clarity. We want to be able to process our pseudo-code mechanically: a preprocessor attached to the Aeolus compiler will convert our pseudo-code to standard Aeolus. This month's report summarizes our ideas regarding this pseudo-code.

The Relationship between our Pseudo-code and Aeolus

Clouds and the Aeolus programming language have been designed in a way which allows the programmer considerable contol over the efficiency of his software. For example, there are six different flavors of objects. Object operations may be invoked as actions in two different ways but need not be invoked as actions at all. A programmer, when invoking an operation on an object or designing an object interface, must be aware of at least eighteen different possibilities. When the various flavors of procedures are also considered this number is even greater.

The intent, by those desiging Aeolus, was to enable a programmer to select from a number of alternative solutions to a programming problem. The programmer, it was hoped, would then be able to select the solution which would work most efficiently in his particular context. The understanding was that different solutions would be appropriate in different contexts.

In developing an Aeolus program it is often desirable to proceed in two steps. First, a programmer should consider the effects he wishes to achieve and the general architecture of his program. Then he should refine that design by selecting particular program structures and flavors of language constructs.

Our pseudo-code is intended to serve as a specification language in that it will allow the programmer to defer a number choices about the structure of his program and the specifics of the language constructs to

be used.

Summary of the Pseudo-code

Our pseudo-code will provide syntax which allows the programmer to delimit the boundaries of actions, to associate attributes with actions (e.g., attributes can be used to declare operations as irreversible or potentially irreversible), and to employ exception handlers. In this section we will summarize our current ideas for the design of our pseudo-code and contrast them with the approaches taken in Aeolus. We will also explain some of the ways in which our pseudo-code may be implemented in terms of Aeolus.

Actions. Our pseudo-code will require the programmer to identify actions statically. An action may be declared as either top-level or nestable. As in Aeolus, a top-level action may be invoked from within another action. A nestable action performs as a nested action if it is invoked from within an action, otherwise it performs as a top-level action. If the programmer wishes to define event handlers, they will be bound to a particular action.

The justification for our approach is that in providing error recovery in the presence of irreversible operations, programmer-defined error recovery will predominate. Our work to date indicates programmer-defined error recovery will be most effective if it sensitive to the semantics of the action in which the fault occured. In providing error recovery which is sensitive to the semantics of a particular action, we want to be able to force the programmer to alway invoke certain operations as an action.

Within Aeolus, actions are created by invoking a procedure or object entry point in a particular way. Thus, an object entry point may variously be invoked as an ordinary procedure, top level action, or nested action. Further, event handlers are bound to the object rather than to the particular action from whose effects recovery may be necessary.

If in binding an event handler to operations we did not also force that operation to be called as an action, we would be making it possible for a programmer to forget to invoke the operation as an action. This error would not be detected except that event handlers for the parent action would be invoked for events within the operation and this could produce recovered states which are incorrect.

Event handlers will be bound to actions even though Aeolus binds event handlers to objects. We will use the attribute mechanism to implement our approach in terms of Aeolus. When an operation is performed on an object a flag (attribute) will be set indicating which operation is in progress. If the action faults, Aeolus will pass control to the event handler for aborts. Since we are using programmer-defined event handlers, it will be constructed to determine first which operation was executing when the fault occured by examining the action in progress flags. Given this information, control will then pass to the portion of the event handler which is sensitive to the semantics of that particular action.

Action Attributes. In the previous section on actions reference was made

to the attribute mechanism. This section describes that mechanism. Attributes are a set of variables which can be used to characterize a particular instance of an action. Attributes should be held in a data area which can be referenced by the action itself; the action's event handler; the action manager, run time system, etc.; and parent actions.

An action's attributes will include any information needed to access the attributes of other actions which were nested within the one described by this particular set of attributes.

We will incorporate the attribute mechanism into our pseudo-code by generalizing on the idea of action type. For an action to have a particular set of attributes, we must declare it to be of the appropriate type. Further more we want programmers to be able to define their own action types.

We will require that an object defininition which is to serve as an action type provide certain standard attributes. This can be done in two ways. The interface between Aeolus programs and the action manager is implemented as a pseudo-object and each instance is already associated with a capability. The action manager is responsible for generating the information about some of the standard attributes. The object containing the attributes for an instance of an action may construct the values for the standard attributes on demand by making appropriate calls on the action mannager. As an alternative, we may construct the values for standard attributes by allowing the action manager to set the standard attribute values by making appropriate calls on the object.

The pseudo-code will allow a programmer to set attributes in the same statement which invokes the action. This will expand to a sequence of Aeolus statements. First, an object corresponding to the appropriate action type will be created. Attributes will then be set appropriately by invoking appropriate operations on the object. Finally the action itself will be invoked.

Exception Handlers and Event Handlers. There are roles for exception handlers and event handlers in our approach to action based programming for embedded systems. When a fault occurs within an action, an exception handler may be invoked to clear the problem and lead to a normal termination. If an action is aborted, (because it faulted and there was not an appropriate exception handler, it committed suicide because it or its exception handler executed an explicit abort statement, or it was aborted by the action manager because a parent action was also being aborted) then recovery is initiated and the abort event handler is invoked. The abort event handler does recovery in two steps. In the first step, some standard tasks are performed (the specifics may be determined by the attributes). In the second step control passes to an appropriate exception handler which has been hand crafted and is sensitive to the semantics of the action being aborted. Under certain conditions both forms of the exception handlers may raise an exception visible to the parent action.

Aeolus provides neither exception handlers nor two stage event handlers for aborts. We beleive we can build both using the programmer-defined event handlers provided by Aeolus. We have already described how event handlers may be crafted so as to be bound to particular actions. We

can also exploit the attribute mechanism to craft these other structures.

Information about whether a particular type of exception has occured will recorded among an action's attributes. The attributes will also indicate whether an explicit abort has been executed. When an exception occurs control must pass to the programmer-defined abort event handler. In Aeolus terms this transfer of control is achieved by aborting the action, though we choose to view it, in terms of our pseudo-code, merely as an exception. The Aeolus level event handler will examine the flags and transfer control to the approriate section of the event handler. The appropriate section may represent an exception handler defined in our pseudo-code. The pseudo-code allows the exception handler to initiate recovery by explicitly aborting the action. This can be implemented as a transfer of control internal to the Aeolus level event handler.

If the flags indicate that the action is being aborted and recovery is required, then control will transfer to that portion of the Aeolus level event handler designated as performing the standard, first step of recovery (in term's of the pseudo-code this is system supplied code which is sensitive to the attributes of the action). While in terms of the pseudo-code the second step of recovery has control transfering to an exception handler. This is implemented via a transfer of control internal to the Aeolus level event handler.

If we wish to raise an exception visible to the parent action, we will set the appropriate attributes. On returning from the nested action, the parent action will be required as a matter of course to examine some flags to determine whether an exception is being raised. If one is, control will pass to an exception handler in the parent action (using the mechanism just described). The code to check the flags and transfer control to the exception handler, if required, will be generated by the pseudo-code processor as part of the action invocation.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

No changes.

4. Summary of Trips and Meetings

Andrew Tanenbaum of Free University, Amsterdam, and Graeme Dixon of the University of Newcastle-Upon-Tyne visited our department during March. They each discussed distributed operating systems research projects in progress within their research groups. Dr. Richard LeBlanc and Tom Wilkes attended the 6th Symposium on Reliability in Distributed Software and Database Systems, along with six other members of the Clouds group. Two papers based on Clouds work, including one co-authored by LeBlanc and Wilkes, were presented.

5. Problems or Areas of Concern

No problems or areas of concern are evident at the current time.

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is sufficient to meet the goals of the contract.

7. Related Accomplishments

None.

8. Plans for Next Period

For Task 1, we plan to do further work on replication techniques for objections and actions. For Task 2, we will continue to develop our ideas about program structures which can be used when implementing one or another of the various flavors of fault tolerance described in the interim report.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	38.5	509.5
Research Scientist II	130.5	783
Grad. Research Asst.	348	2741
Secretary	40	475
Clerk Typist	47	475

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: April, 1987

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Task 1 (Programming Techniques for Resilience and Availability):

In our last monthly report, we described our preliminary work on the theoretical and technical framework of a method for deriving a replicated version of an object from its single-site implementation. In this report, we describe this framework, called *Distributed Locking*, in greater detail.

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. Rather, it allows the user to specify policy in the areas of replica concurrency control (*e.g.*, the quorum consensus or available-copy algorithms) and state copying (*e.g.*, idemexecution or cloning). The motivation to provide this flexibility derives from study of the many proposals for replication control that have appeared in the literature in recent years. It has become clear from the wide diversity of these proposals that tradeoffs between availability and consistency of replicas are not only possible, but in some applications highly desirable. Thus, in order to provide support for replication in a general manner, it is essential that the user be allowed to take advantage of semantic knowledge to tailor the replication control in an application-specific manner. The mechanism provided by Distributed Locking for this purpose is described below.

Distributed Locking, however, also allows the user to specify that one of several default (pre-programmed) strategies be used in each of the two areas of replication control (*e.g.*, the combination of quorum consensus and cloning). Thus, in accord with the general philosophy of such mechanisms in Aeolus/Clouds—as demonstrated in its features for control of synchronization and recovery—provision is made both for automatic provision of replication control, and for "roll-your-own" specification of control by the user when desirable.

Distributed Locking: Methodology

The basis of the Distributed Locking methodology was described in our last monthly report. Essentially, derivation of a replicated object from its single-site implementation consists 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.

Note that, as discussed in our last monthly report, 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.

Policies for control of concurrency among replicas, and for control of the copying of state among replicas, are expressed in the *lock* object event handler and the *copy* action event handler, respectively. Preprogrammed default handlers for these events, which implement commonly-used schemes such as those mentioned above, 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, 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 accomodate 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).

Distributed Locking: Mechanism

The mechanism required for support of Distributed Locking requires the modifications to the Clouds object and action naming schemes which have been proposed for the support of the Parallel Execution Thread (PET) scheme, as described in our last monthly report. The Distributed Locking mechanism also requires modification of the Aeolus/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. (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.

Note that, when a lock is obtained at an s-cohort, the s-cohort is automatically added to the *touched list* for the controlling action. Thus, when the controlling action commits at the p-cohort, normal commit processing occurs at each of its s-cohorts as well. This is useful, for instance, when state cloning via copying of shadow sets is used; the shadows are committed at each of the s-cohorts as if the shadows had been produced by execution at that s-cohort.

Task 2 (Action-Based Programming for Embedded Systems):

We are developing several examples illustrating the recovery mechanism we have designed. Last month, we described the pseudo-code being used to develop these examples. This month we will outline the examples and explore some system architectures which depend directly on our recovery mechanism. The mechanism we have proposed allows for both forward and backward recovery, and integrates the handling of aborts with the handling of exceptions in general.

As our work has progressed, we have developed a more generalized understanding of the applicability of our approach. Initially, we concentrated on the problem of recovering actions which had performed irreversible operations. More general versions of this problem arise when we consider the issues associated with recovery in systems where atomicity is not strictly enforced and in systems where the state of the physical system in which the software is embedded is not completely under software control. While we began our work believing that irreversible operations were a problem to be overcome, it now appears that software structure may actually be made more robust if a programmer is allowed to treat certain

operations as irreversible. Our examples will include one which considers a way in which the idea of an irreversible operation may be used to advantage.

It is our belief that the judicious use of a recovery mechanism such as we are developing can simplify the structure of the software system by reducing the need to propagate control information among actions. Simply put, in our model an action is responsible for cleaning up after itself even if it fails. This makes actions more self contained. A mechanism is provided which allows a programmer to separate the mainline of the action from the various provisions for doing the clean up. This facilitates the process of constructing actions which are indeed self contained.

The importance of actions being self contained is evident in a system such as Clouds in which a program's execution is expected to thread its way through many persistent objects which are shared among a number of different processes. An action may be invoked into a variety of different environments and in each the programmer must defend against incomplete or erroneous results. If an action is complex or contains a large number of points at which errors may occur, a programmer is likely to be ineffective in his use of defensive programming techniques. The use of a recovery mechanism to make the actions self contained makes it possible to push the burden of defensive programming on to the one who defines the action rather than the one who uses it.

In this report, four examples are summarized. The first is a revised version of the Information Processing example introduced in the interim report. This example is developed to illustrate an architecture in which certain operations are designed to be irreversible even when this is not strictly necessary.

The second example extends the first one in order to illustrate how the recovery mechanism can be used to support hardware and software maintenance. The example suggests it may be possible to maintain a system without interrupting its operation.

The third considers how our recovery mechanism might facilitate the switch over to a backup system.

The fourth example illustrates how our recovery mechanism might be used in an embedded system which periodically checks its assumptions about the state of the system it is controlling against sensor data.

Example 1. A "server" is a rather generally employed software entity. It is a program which operates or manages some designated system resource. This example considers software designs in which the process using the server must have exclusive access to it, e.g., a printer. There are two rather commonly used strategies by which processes may share access to it: processes may block until they get their turn at the server; or requests for service may be queued, and process never blocks. In the case of the printer, this second strategy is frequently called "spooling."

Because of the timing constraints which must often be satisfied by embedded systems, it is better to queue requests than to allow processes to block. The strategy of queuing requests for service and not allowing the process to block works if the request is guaranteed to be serviced. In terms of our work here, the strategy works as long as the request for service is irreversible.

While we will develop our ideas in terms of a printer server, they should generalize easily to the construction of other servers for resources requiring exclusive access.

This example will explore a software architecture which allows us to guarantee that print requests will be honored. We do this by constructing the print request so that it employs various recovery techniques to ensure that it is not destroyed. This allows actions to commit and processing to continue even if there is heavy backlog

of print requests.

The material to printed will be encapsulated within an object we will call a print object. The capability for the print object will be placed in a queue known to the printer server. When a print object is to be granted access to the printer, the server invokes an appropriate entry point on the object. This entry point is an action. The print object then constructs the text to be printed and passes it on to the server by invoking server entry points. The print object can be thought of as containing a protocol for operating the printer. If there is a failure within the print action (in the print object), perhaps one of the operations in the protocol for operating the printer aborted, the recovery mechanism within the print action will attempt recovery, e.g, by sending itself to another printer. Under such a circumstance the print action may abort and raise an exception visible to the server. As part of handling the exception, the current instance of the server will terminate but should first start another instance of a server (based perhaps on a different version of its code.)

If the action which placed the printer request in the first place should subsequently fail at least two courses of action are possible. Recovery within that action could merely set a flag indicating the report had already been printed. This would be done in anticipation that the action would be retried. We are working a a scheme by which an action can initiate its own retry as part of its recovery process. Even though the print action was irreversible, it may be possible to prevent the report from being printed through the use of appropriate compensatory operations.

Example 2. The second example will illustrate how the recovery mechanism can be used to facilitate software maintenance.

If a printer fails and is replaced with a different device, it may be necessary to install a different device driver and this in turn may force adjustments in software throughout the system. With a recovery mechanism, it would be possible to make most of the adjustments automatically. When the device driver discovers that the printer has been changed (perhaps by reading a register containing a printer identification number), the device driver aborts, the recovery mechanism creates an object containing the correct driver, transfers state information from the old driver object, and then restarts the server process.

This idea can be extended to problems deriving from incompatible versions of objects as well. Suppose the print action (in the print object) attempts to use an out of date protocol. It should abort. As part of recovery the print object should be recast using the current version. While this ensures that the report gets printed, some additional work might be required to track down the source of the old version of the print object. There might be a copy of the create print object operation in a place unknown to software maintenance (perhaps some undocumented replication, or perhaps a loophole in the code which allowed programs to employ short cuts when creating print objects).

Example 3. In this example we will consider the problem of achieving fault tolerance by cutting over to backup systems. The example we are developing will involve switches between automatic and manual control.

The switch from automatic to manual control will be triggered because the physical system being controlled enters a state which was not anticipated by the software designers. An operator must be alerted and provided with information about the state of the system. The operator will use his controls to return the system to a state from which automatic control can be resumed. If the operator does not intervene effectively, there will be a second cutover to a system which

will shut the equipment down safely.

The underlying idea is that the mechanism for cutting over to a back up system is essentially the same regardless of the particulars of the event triggering the cut over. The cut over involves creating an appropriate set of objects and establishing their states. There will be objects which are common to both the primary and backup mechanisms and it will be necessary to properly link them with the objects specific to the backup system. While there is an alternative approach which would have the back up mechanism always in a state of readiness, the dynamic relinking of objects would still be required.

This example will be developed in terms of a software system which controls an electro-mechanical process. The principles illustrated will be appropriate for controlling manufacturing systems as well as weapons systems.

Example 4. Some of the most difficult problems in constructing embedded systems are associated with feedback loops. In example four, we will several problems associated with validating and utilizing information obtained from sensors to adjust equipment. In particular we will focus on the problems associated with constructing systems where there is considerable delay in obtaining reliable feedback about the state of the system. Our example will explore a programming strategy which requires the software to makes its "best guess" about adjusting a system (employing mathematical models). If the feedback suggests the guesses were acceptable, the action will continue. If it turns out based on feedback that the guesses were not very good, the action will abort. As part of the recovery process, the parameters in the model will be adjusted. This is complicated because even if the action is restarted the initial conditions may have changed. We have not yet convinced ourselves that programs of this sort will be easier to conceptualize and write if we employ an explicit recovery mechanism. Our example will be developed in terms of a system in which there are several logically distinct levels of feedback.

2. *Special Programs Developed and/or Equipment Purchased*

None.

3. *Key Personnel*

No changes.

4. *Summary of Trips and Meetings*

None

5. *Problems or Areas of Concern*

No problems or areas of concern are evident at the current time.

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is sufficient to meet the goals of the contract.

7. Related Accomplishments

None.

8. Plans for Next Period

For Task 1, we plan to do further work on availability specifications, developing syntax for incorporating them in Aeolus programs. For Task 2, we will continue to develop the examples presented in this report.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	81	527.5
Research Scientist II	130.5	913.5
Grad. Research Asst.	348	3,089
Secretary	47	522
Clerk Typist	40	515

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: May, 1987

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Task 1 (Programming Techniques for Resilience and Availability):

In our previous monthly report, we described the *Distributed Locking* scheme for deriving a replicated object implementation from a single-site specification. As outlined then, the scheme consists of a methodology for achieving this derivation together with a mechanism to support it. The mechanism consists in part of a set of primitives with which both system-provided (default) and user-specified *lock* and *copy* event handlers may be programmed. Here, we will describe these primitives in greater detail.

The interface to these primitives is provided as an Aeolus pseudo-object, called *DistLock*. This pseudo-object is imported automatically by every *availability specification* (availspec), but is not available for import by other compilands. This restriction is made to prevent use of the primitives outside of an availspec, because most of the primitives make the assumption that they are invoked within the environment of a *lock* or *copy* event handler.

The pseudo-object defines a *replica_number* type which is used by most of the primitives:

```
type replica_number is new integer
```

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 our interim report. 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.

The first primitive is used for propagation of a lock to one of a set of replicas of the invoking object:

```
procedure lock_replica ( rep : replica_number ) 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 *invoke_replica* primitive is used for implementing state copying by idemexecution:

```
procedure invoke_replica ( rep : replica_number ) modifies
```

This 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 *broadcast_shadow* primitive is used for implementing state copying by cloning using shadows:

```
procedure broadcast_shadows () modifies
```

This 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. This operation should be invoked only within a *copy* event handler.

Finally, we define an additional object event, called the *accept event*, which is used by a given replica to transfer a user-specified portion of its state to another of the replica's group. This event must be explicitly signalled by the user via the *invoke_acceptor* primitive. This primitive 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 invoke_acceptor ( rep   : replica_number ,
                            state : address        ,
                            len   : longuns        ) modifies
```

This 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*.

These and other implementation details of the Distributed Locking scheme are currently being developed in a dissertation by one of the researchers on this grant. In our next report, we will describe the syntactical features of **availspecs** in greater detail, and will also present results of our attempts to derive higher-level linguistic features for specification of resilient objects from our experience with the lower-level features provided by Aeolus.

Task 2 (Action-Based Programming for Embedded Systems):

This status report begins a consideration of the problems related to using a forward recovery mechanism in a disciplined way. The examples we worked on suggest there are a variety of ways in which the forward recovery mechanism may be used to tolerate hardware and software faults. The software designer must make some choices early in the development of the system about which strategies will be used to achieve fault tolerance. These strategies in turn constrain the structure of the system and the ways in which objects may interact. Without such constraints guiding the design of the mechanisms for achieving fault tolerance within objects and actions, attempts at forward recovery are unlikely to correct or compensate for a fault effectively, and may, in the worst case, induce a cascade of faults throughout the system.

The variety of strategies available is indicated in terms of a particular type of fault: dangling capabilities. We also outline some of the questions which must be resolved if we are to build well designed systems which incorporate effective fault tolerance mechanisms.

In the Clouds environment, much of the code for systems and applications will be encapsulated within objects. An important class of irreversible operations involves the management of these objects. An object will, in general, be referenced by several others objects, i.e., it will be shared. As with any object it is possible to delete a shared object. The deletion of an object, as with other less severe state changes, may well be regarded as irreversible. The complication which arises when shared objects are deleted is that of dangling capabilities: it is necessary to reestablish consistency within the collection of objects by propagating information about the deletion throughout the collection. There are several ways in which information about the deletion may be propagated through out system. On one level, these techniques separate into two broad classes: those which can be employed at the time of the deletion or other state change and those which may be employed when an action attempts to use the dangling capability. On another level, however, these two classes of mechanisims use many of the same strategies and may actually be used in tandem.

An object may attempt to keep a list of other objects which reference it. The objects named in this list may then be notified when the object is deleted. There are a number of alternatives as to the form this notification may take including that of invoking the recovery managers in the referencing objects.

Not all objects may maintain a list of the objects referencing it. Furthermore, there is no practical way of guaranteeing that all objects in possession of the deleted object's capability will have been properly registered. Thus, there remains a need to cope with actions which attempt to use dangling capabilities. When an action attempts to use a capability left dangling by a delete operation an exception should be raised and the action may be aborted.

What strategies are available for reestablishing consistency? In some contexts, it may be reasonable to suppose that the deleted object has been replaced with a successor. In other contexts, it may be more reasonable for the action with the dangling capability to recognize the object it's attempting to access no longer exists and to redirect its computation down another path.

If object O1 is to be deleted it may actually be more reasonable to redefine it instead. Suppose an action A1 is executing in object O2 and while doing so executes one of O1's entry points. The redefined operation could raise an exception indicating that O1 is no longer a valid object an initiating appropriate recovery within A1. If O1's deletion was part of a system upgrade, then the redefined O1 may be even more sophisticated: it may include information about a temporary fix which allows A1 to perform its task until it too is upgraded, or O1 may even include information about how A1 may participate in its own upgrading. In this later approach, A1 may dynamically rebind some of the operations in O2 or may notify an operator that maintenance is required.

Within Clouds, there are two ways operations may be dynamically rebound. First, operations which at a higher level may appear to be encapsulated within a single object may in fact be partioned among several objects and invoked indirectly. The rebinding of operations, then, may be accomplished by using a different capability when accessing the code which has been partioned off and encapsulated within a separate object.

A second and more general approach to the rebinding of objects is made possible through the use of windows among objects' address spaces. While each Clouds object is thought of as existing within its own address space, the Clouds design allows for portions of one object's address space to be mapped into the address space of another object. While this feature has not been implemented within the current Clouds prototype, the appropriate hooks have been provided.

There are three questions which must be followed up in the short term. First, how might the sharing and dynamic rebinding of operations and data areas be used to support the general task of maintaining consistency within a set of interdependent, persistent objects.

Second, the forward recovery techniques we have described raise some new difficulties related to the coordination of actions executing concurrently within the same object. In particular, what are the implications of performing recovery on an object (either by invoking an actions abort handler or the recovery manager within the object) while a second action is executing within the object? We believe that if the second action holds locks on data areas to be accessed during recovery, the second action should be aborted before recovery proceeds. This requires some additional consideration, however.

Third, the forward recovery mechanism may, if used in an undisciplined way, result in "sloshing." For example, suppose that A1 aborts, recovers and restarts itself, though in the process sets up conditions causing A2 to abort. If A2 recoverys and restarts it may in turn set up conditions causing A1 to abort. Unless precautions are taken, this cycle may continue without end. We must, then, look for a structured way in which to use forward recovery which allows us to avoid the "sloshing" effect or to at least stop it if it occurs. Our working conjecture is that the solution is to be found in structuring the connections among objects appropriately.

2. *Special Programs Developed and/or Equipment Purchased*

None.

3. *Key Personnel*

No changes.

4. Summary of Trips and Meetings

None

5. Problems or Areas of Concern

No problems or areas of concern are evident at the current time.

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is sufficient to meet the goals of the contract.

7. Related Accomplishments

Notice has been received that we have been awarded a Coordinated Experimental Research grant from NSF. This grant will provide substantial hardware and personnel resources for further development of the Clouds testbed over the next five years.

8. Plans for Next Period

For Task 1, we plan to continue our work on availability specifications. For Task 2, we will continue our development of recovery techniques, concentrating on the problems listed in the progress report.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	81	608.5
Research Scientist II	130.5	1044
Grad. Research Asst.	348	3,437
Secretary	40	562
Clerk Typist	47	565

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: June, 1987

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Task 1 (Programming Techniques for Resilience and Availability):

In our previous monthly report, we described primitives for use in implementing the `lock` and `copy` event handlers required by the Distributed Locking scheme. In this report, we present example event handlers demonstrating the use of these primitives, as well as an example availability specification (`availspec`) for the `symtab` object making use of the example event handlers. (The Aeolus source code for the recoverable `symtab` object was presented and described in our interim report.)

```
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.

   num_locked   : integer
   this_version,
   max_version  : version_number
   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 1. Lock and Copy Event Handlers for Quorum Consensus

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 1. 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 previous reports, a 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.

Example of an Availability Specification

A sample `availspec` making use of the `quorum` event handlers is given in Figure 2. This `availspec` applies to the resilient symbol table object which was presented and described in our interim report. 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) to the `Create_Instance` operation of the `TypeTemplate` for this object.

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.

Task 2 (Action-Based Programming for Embedded Systems):

In Clouds, software reuse is achieved by means of persistent, shared objects. It is important for pragmatic reasons that objects with similar purpose have similar interfaces and employ similar strategies to achieve fault tolerance. In this report we introduce the possibility of using class hierarchies and inheritance schemes to help ensure that faults are

```
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 2. Availability Specification for the Resilient Symbol Table

handled in a consistent manner at various points within an application or class of objects.

We have been studying the use of forward recovery mechanisms to achieve forward progress in the face of faults. We have associated forward recovery with actions. If an action is aborted or faults control is transferred to the recovery handler associated with the action. The recovery handler may be used to undo the effects of the action or, if not all of the action's effects can be undone, the recovery handler may be used to establish consistency among the objects and other data visible to the action.

An action will generally modify several objects in the course of its execution. Some of these objects will have been modified directly by the action. Others will have been modified indirectly by means of nested actions. Suppose A is an action executing within an object O1. Suppose further that E is an action nested within A. E is an entry point to object O2. E

makes changes to the state of O2 and also invokes entry points on O3 and O4. The entry points modify the states of O3 and O4 but do not access any other objects. Action A has modified O1 directly and O2, O3 and O4 indirectly. Suppose now that action A subsequently is aborted and must recover.

The simplest strategy would be to use backward recovery and return all the touched objects to their states as they were prior to the invocation of action A. If this is not possible, a forward recovery scheme will have been supplied for action A. There are at least three possible scenarios by which consistency can be established with respect to O1, O2, O3, an O4. The recovery handler could modify the state of O1 directly and invoke additional entry points on O2 so as to return it to a consistent state. In this scenario the recovery handler would depend on the semantics of the entry points of O2 to make any necessary changes to the states of O3 and O4. The second scenario is similar. O2 could be equiped with a special entry point called RECOVERY. Recovery in this case the recovery handler could modify the state of O1 directly and then invoke O2@RECOVERY(). The states of O3 and O4 would be recovered indirectly by means of the recovery operation invoked in O2. The third scenario arises if the designer of the recovery manager for A wishes to achieve more direct control over the recovery of objects indirectly modified by A. In this scenario the runtime system would provide the recovery handler with a list of objects touched by A. The recovery handler would then take steps appropriate for each object appearing in the touch list.

Additional scenarios can arise if we consider the ways in which O1's state can be segmented into a number of recoverable data areas or if the semantics of A required that the recovery handler invoke entry points to objects not previously touched by A. Perhaps the most important scenarios arise by allowing the recovery handler to be sensitive to the history of operations performed on an object by the aborted action: for example, a sequence of operations could be undone by invoking the inverse for each of the operations in the sequence.

The important point is that there are a number of strategies by which recovery can be attempted. If an action attempts to use one strategy on objects which were designed to be recovered by a different strategy, the attempt at recovery will fail. Since capabilities to objects can be passed around as arguments, it is important that consistency be maintained. Suppose an action A receives the capability to an object X as an argument. How should A attempt to recover X in the event A aborts? Whatever strategy is chosen, it must be appropriate for the entire class of objects which can be bound to X. We believe that we can develop a class hierarchy which allows us to associate recovery strategies with classes of objects.

An object will inherit methods (e.g., entry points and recovery strategies) from its parent class. A programmer will have the option of extending, modifying or overriding any of the inherited methods when instantiating a particular object. The inheritance mechanism is not used to resolve the meaning of an operation invoked on a particular object. Instead, we are investigating the use of inheritance to generate a code template which the programmer is free to edit in any way he may regard as appropriate.

When overriding an object's inherited interface (e.g., deleting an entry point or changing a parameter list) or when restructuring an inherited recovery strategy, the programmer is responsible for ensuring that the object is structured in a way that is compatible with the way other objects will be referencing it. The code template generated by the inheritance mechanism has a very special role to play: it indicates to the designer of an object (or object type) the structure that object is expected to have by other objects within the system.

We also plan to explore the possibility of developing a multiple inheritance scheme to facilitate the editing of code templates. An object's definition can be regarded as a set of traits. Traits may themselves be defined in terms of a class hierarchy with inheritance. A designer should edit an object's definition by adding and deleting the appropriate traits. If no

mix of traits provides exactly the semantics required for the object, the designer should first define the appropriate trait within the trait hierarchy and then edit it into the object's definition. This ensures that information about the particulars as to how this new object must be manipulated are available to the designers of other objects which will be referencing it.

In last month's report we discussed some of the difficulties which a programmer must confront if forward recovery techniques are to be used effectively. On reflection, the central problem appears to be one of ensuring that the programmer has detailed information about how his actions and objects will interact with the variety of persistent, shared objects available within his object space. In this month's report we have outlined an approach intended to provide the programmer with that detailed information. Indeed, we believe the approach we have outlined will provide consistency of structure across an object space. In particular, we belive the approach will help ensure that objects with similar purpose whave similar interfaces and employ similar strategies to achieve fault tolerance. Inconsistencies, when they occur, will be the result of a programmer's deliberate choice rather than a lack of information. The inconsistencies will themselves be documented so that the information will be available to other programmers.

2. *Special Programs Developed and/or Equipment Purchased*

None.

3. *Key Personnel*

No changes.

4. *Summary of Trips and Meetings*

Richard LeBlanc travelled to RADC for an interim review of research progress.

5. *Problems or Areas of Concern*

No problems or areas of concern are evident at the current time.

6. *Sufficiency of Effort Toward Meeting Goals of the Contract*

The current level of effort is sufficient to meet the goals of the contract.

7. *Related Accomplishments*

None.

8. *Plans for Next Period*

For Task 1, we will be developing the concept of "resilient types," a declarative approach to efficient use of atomic actions. For Task 2, we will be formulating a strategy to develop examples illustrating the use of class hierarchies, inheritance, and traits to facilitate the definition of new objects.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	81	689.5
Research Scientist II	130.5	1,1174.5
Grad. Research Asst.	348	3,885
Secretary	47	609
Clerk Typist	40	609

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: July, 1987

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Task 1 (Programming Techniques for Resilience and Availability):

In this report, we describe a proposal for a declarative language feature called the *resilient type*. This new feature embodies the paradigm we have developed from our experience with the more imperative features for resilience described in previous reports, *i.e.*, per-action and permanent variables.

The typical use of per-action variables is for "intention lists" of modifications which are to be applied to permanent variables during action commit. When using per-action and permanent variables to achieve resilience of permanent data, the programmer must specify the following characteristics:

1. the representation of the permanent version of the data;
2. the relationship of the `modify` operations of the object to the permanent representation; and
3. 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 `modify` 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, `modify` 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.

We have developed a preliminary design 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 Figure 1. This example is derived from the resilient symbol table object that was presented and described in our interim report. 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 representation of the permanent data structure may be accessed within the resilient type specification by the name `rep`. 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. Other details of the prototype syntax are given in the comments contained in the figure.

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. The

Aeolus language does not support iterators; thus, we do not recommend that the resilient type construct be included in Aeolus itself, but rather in an application language for Clouds which should include such high-level features as iterators.

```
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.
```

Figure 1. Symbol Table Example using Resilient Type

Task 2 (Action-Based Programming for Embedded Systems):

The approach to forward recovery we have been developing places the locus of control (for recovery) with the abort handler associated with the action being aborted. The abort handler will execute a protocol to recover (i.e., restore consistency to) the data areas and objects visible to the aborting action. The protocol must be compatible with the objects and other data areas being recovered. Since actions and objects are defined at different times by different programmers, we must take steps to ensure that recovery protocols are compatible with object definitions. Our idea is that the programmer who designs an object should make available to the user of the object code templates which illustrate proper recovery. The programmer defining an action can incorporate that template into his code after tailoring it to the particular context of the action he is developing.

Additional constraints must be placed on the programmer designing an object: like objects should be recovered using similar protocols. Suppose there are two resources R and S. Both R and S are encapsulated within objects. Suppose further that an action A may use either R or S (and does not care which). The abort handler will be simpler if R and S can be recovered using the same protocol. To ensure that this is the case, the definitions of R and S must be coordinated in some fashion. As discussed last month, we are exploring the use of class hierarchies and inheritance schemes to provide the necessary coordination. Our proposal differs from others in that the class hierarchies and inheritance schemes are used to generate a code template which the programmer is free to edit--- in short, they are used to produce a "rough draft" of the code. The class hierarchies and inheritance schemes are not involved in the execution of the software; they are merely aids to help programmers coordinate the structure of the software system.

To develop this idea further we return to the example of a resource manager discussed in a previous report. The resource manager M is an object which controls access to a pool of resources. Actions may check resources out of the pool as needed, but the resource must be returned when the action is finished with it. An action checks a resource out with the operation M@Request(X) and returns it with the operation M@Return(X). If an action

holding a resource aborts, it must, in general, return the resource before it terminates. Thus, the abort handler for an action which checks a resource out from M will usually contain the M@Return(X) operation. A programmer may sometimes wish to dispose of the resource in other ways: an orderly way of allowing for exceptions is discused later in this report.

Classes and multiple inheritance can be used to construct appropriate code templates. One approach would be to have one class hierarchy for objects and another for actions. The resource manager M would be a member of a class of objects called "ResourceManager." The idea being that all
resource managers will have similar interfaces for checking resources out and returning them. The programmer defining the class ResourceManager would also define an attribute for actions which access a ResourceManager called "ResourceUser." When an action is identified with the attribute ResourceUser, the appropriate code for returning a resource to the pool is incorporated into the code template for the action.

More sophisticated schemes should be available for those occasions when a programmer does not wish a particular action to return a resource during recovery. We will mention three strategies here.

1. The programmer may specify that the action recover and then be retried some number of times. It may be desirable as part of ensuring forward progress for the action to hold onto the resource between retries.

2. The programmer may specify that the action recover by starting a new action and transferring control to it.
In addition to transferring control, some state information may also be transferred including the identity of resources held by the aborting action.

3. The programmer may specify that the action recover by passing the resources to its parent before terminating.

These more sophisticated schemes should be defined as subclasses of the ResourceUser attribute. If a programmer is defining a class of actions A and if this class is using a resource whose manager's object type is ResourceManager, he will want the definition of A to contain an appropriate recovery protocol, i.e., an attribute in the ResourceUser family. This may be the ResourceUser attribute itself or one of its more sophisticated subclasses such as those mentioned above.

Our approach to the construction of objects and actions asks a programmer to generate a code template for the program unit by selecting an appropriate mix of attributes. The correct choice of attributes depends on the purpose served by the program unit, and by the attributes of the other program units which it references or which reference it. This approach should facilitate the correct and consistent use of recovery protocols by providing the programmer with access to code fragments which solve his problems. In addition, the programmer will need some guidance as to the correct use of attributes. We propose that this guidance be provided by means of constraints. One constraint we have discussed is that of "an action which may obtain a resource from a ResourceManager must have an attribute in the ResourceUser family." Other constraints would be attributes in the ResourceUser family. For example, if an action passes a resource to its parent when it aborts (we'll call this the "LeaveToParent" attribute), the parent must be prepared to handle it; this can be expressed by placing constraints on the attributes assigned to the parent of an action with the LeaveToParent attribute.

To summarize: We propose the use of object classes and action classes. A class is defined by combining a set of attributes. A class may have subclasses. A subclass initially inherits its parent's attributes, though a programmer may subsequently add and subtract attributes (to distinguish it from its parent). Attributes are associated with code fragments and are defined as a combination of other attributes. As with actions and object classes, attributes are organized hierarchically. Code templates are instances of classes. Once a code

template is generated from the definition of its class, the programmer is free to edit it as appropriate.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

No changes.

4. Summary of Trips and Meetings

Richard LeBlanc attended the Tenth Minnowbrook Workshop, the topic of which was software reuse. A number of the participants were involved with the development of real-time software. Their presentations provided us with valuable insights concerning how the results of our research might be used.

5. Problems or Areas of Concern

No problems or areas of concern are evident at the current time.

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is sufficient to meet the goals of the contract.

7. Related Accomplishments

None.

8. Plans for Next Period

For Task 1, we will be working on the organization of the hardbook to be delivered as the result of the task. For Task 2, we are considering the use of these ideas regarding class hierarchies and inheritance schemes as a vehicle for organizing the handbook which will be produced at the end of this task. We will consider the structure of the proposed hierarchies, the notation for representing code fragments, and the mechanism for generating code templates from the definition of a class.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	117	806.5
Research Scientist II	130.5	1,305
Grad. Research Asst.	348	4,233
Secretary/Clerical	87	1,305

G-36-645

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: August, 1987

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Task 1 (Programming Techniques for Resilience and Availability):

Outline for Guidebook 1

I. Taxonomy of Techniques for Programming Resilient and Available Services
  1. Resilience
    a) Autorecoverable
    b) Recoverable
    c) Per-action and permanent variables
    d) Resilient types
    3) K-resilient computations (ISIS)
    f) Mutex types (ARGUS)

  2. Availability
    a) Ad-hoc techniques
      1) Master/Slave
    b) Distributed Locking
      1) Primary copy
      2) Token passing scheme
      3) General quorum consensus
      4) Available copies

II. OPERATING SYSTEM REQUIREMENTS
  1. Support for resilience
    a) Action Management
    b) Object Headers

  2. Distributed Locking Mechanism
    a) Distributed Locking Mechanism
      1) Naming Replicated Objects
      2) Invocation of Lock and Copy Events
      3) Primitives for Lock and Copy Event Handlers
    b) Object Filing System

Task 2 (Action-Based Programming for Embedded Systems):

The following material summarizes our work so far and is meant to provide a basis for the organization of the handbook resulting from this task.

I. The stages of fault tolerance recovery

(a) detect the presence of a fault: different types of faults require different strategies
(b) confine the consequences of the fault
(c) adjust the state and resume computation

We have concentrated on the last stage. We have regarded the issue of confining the consequences of a fault in its simplest terms: preventing the failure of an action from shutting a software system down and ensuring that all actions have a consistent view of the state of the computation when recovery is complete. When framed in this way, the distinction between the last two stages blurs. We believe that the use of *atomic* actions is an important strategy in containing the consequences of a fault. Unfortunately, there are

many occasions when atomicity cannot be maintained. Our emphasis has been on studying the use of forward recovery in situations where atomicity has not been maintained. We have assumed that these "holes" in atomicity have been designed with the containment of faults in mind. We will not discuss recovery in situations where the fault has been allowed to contaminate the rest of the computation in arbitrary ways.

Violations of atomicity may make an action *irreversible*. We regard as irreversible any action whose effects cannot be undone merely by rolling back the data areas and objects on which it has write locks. Irreversible actions may arise in two ways. An action which causes a state change in a physical system is termed irreversible. The effect of the state change is visible even before the action commits. The use of backward recovery (e.g., rollback) in this case risks the loss of information about the occurrence of the operation on the physical system. Violations of atomicity must also sometimes be allowed to provide the necessary level of availability (e.g., a resource manager). In this case, backward recovery will either trigger a cascade of aborts or result in actions having inconsistent views of the state of the object.

These two types of irreversible action are closely related, and we have been studying how their recovery may be addressed using forward recovery techniques. Forward recovery can be used to ensure that information is preserved across the boundary of the aborting action and is available when computation is resumed. In particular, it can be used to preserve information about the occurrence of operations on physical systems under software control and to preserve that portion of the state which has been seen by another action. In this later case, reestablishing consistency may require that some additional, selected actions be aborted and restarted on the adjusted state.

II. Philosophy and Issues

We believe an abort handler should not be flooded with information about the state of the computation as it was at the time of an abort. It should be given only the information it asks for and only in the level of detail it requires. To this end, we have developed a forward recovery mechanism which incorporates ***stages*** and ***diagnostic operations***.

We want to work with a level of granularity between that of the action as a whole and that of individual statements. The programmer is able to organize the action in terms of ***stages***. A stage cannot contain part of statement: a stage boundary cannot be within the scope of a loop or conditional.

The abort handler will use additional diagnostic operations if it needs more information about the conditions which existed at the time of the abort. Some diagnostic operations will be provided as system calls while other will be object entry points.

Some of the diagnostic operations will tell the abort handler about the reasons the action aborted. Others will tell the abort handler the state of an object at the time of the abort. Yet others will tell the abort handler whether a particular operation was performed before the abort.

III. The outcomes possible using forward recovery
(a) the list from the interim report

IV. Some generic capabilities
(a) adjusting environments and data areas

i. Environments within a block may be partitioned into data areas. these data areas may be selectively committed or rolled back. The abort handler may make changes to a data area before committing it.

ii. An environment may contain capabilities for other objects. If an action has modified an object and then aborts, the abort handler may have to initiate recovery in the object. The precise way recovery is to be handled for a particular object depends on the semantics of the object. We require that the object's entry points include the operations needed for recovery. This is especially important if operations on the object are irreversible or otherwise allow the action's atomicity to be violated. If an action has exclusive access to an object, then the object may be treated in the same way as data areas, i.e., rolled back and committed. We assume the programmer of an action knows the semantics of the objects and have correctly provided for their recovery. This is not a good assumption, and we are developing some concepts for tools which will help a programmer provide for recovery correctly.

(b) transfers of control

i. the abort handler terminates the action after propagating some of its local state into the global environment. The abort handler lets control pass to the parent action either by raising an exception or terminating normally.

ii. the abort handler terminates the action after propagating some of its local state into the global environment and starting a new action to continue the computation. The new action has the same parent action as the one which was aborted. The new action executes in an environment defined by static scope rules.

iii. the abort handler restarts the action after making some adjustments to both the action's local and global environment.

iv. the abort handler restarts the action in an intermediate state after making some adjustments to both the action's local and global environment.

v. the abort handler remaps the code windows (and perhaps data windows) of the action and then adjusts its local and global environment. The action is restarted either from the beginning or from an intermediate state. The new action executes in the same environment as the one which had aborted.

vi. some recovery may be deferred until the action is restarted.

vii. some actions may be given the right to abort selected other actions, thereby forcing them to begin (forward) recovery

V. Some scenarios of interest utilizing the generic capabilities

(a) some variations on the resource user

(b) avoiding cascading aborts by restarting an action in an intermediate state

(c) the fire control action

(d) the parent of the fire control action

(e) shutting down a machine tool using the remapping of code windows

(f) robotics control example

(g) restarting an action on another machine

(h) propagating information about physical systems across machines

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

No changes.

4. Summary of Trips and Meetings

None.

5. Problems or Areas of Concern

No problems or areas of concern are evident at the current time.

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is sufficient to meet the goals of the contract.

7. Related Accomplishments

Tom Wilkes completed his Ph.D. thesis, which included a substantial amount of material from Task 1 of this project.

8. Plans for Next Period

For Task 1, we will begin to draft the handbook to be delivered as the result of the task. For Task 2, we will be working on examples utilizing the capabilities discussed in this report.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	117	923.5
Research Scientist II	130.5	1,435.5
Grad. Research Asst.	348	4,581
Secretary/Clerical	87	1,392

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: September, 1987

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Task 1 (Programming Techniques for Resilience and Availability):

Work has begun on a draft of the guidebook to be delivered as a result of this task.

Task 2 (Action-Based Programming for Embedded Systems):

We have been developing a set of skeletal examples to illustrate and study the effectiveness of the generic recovery capabilities listed in the last monthly report. Several of these examples are attached to this report as Appendix A.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

Tom Wilkes and Chu-Chung Lin departed from Georgia Tech and the project staff during this month. One new graduate student, Ray Chen has been added to our staff.

4. Summary of Trips and Meetings

None.

5. Problems or Areas of Concern

No problems or areas of concern are evident at the current time.

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is sufficient to meet the goals of the contract.

7. Related Accomplishments

Chu-Chung Lin completed his Ph.D. thesis, on the design of debugging tools for object/action programs.

8. Plans for Next Period

For Task 1, we will continue our work on the handbook. For Task 2, more examples will be developed to complete our analysis of the capabilities described in the August monthly report.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	117	1,040.5
Research Scientist II	130.5	1,561
Grad. Research Asst.	348	4,929
Secretary/Clerical	87	1,479

Skeleton of an action illustrating the basic template

```
!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
```

Designing Fault Tolerant Resource Users

Suppose an a pool of resources is encapsulated within an object called the ResourceManager. Actions may check resources out of the pool by invoking the appropriate entry point in the object. Access to the ResourceManager represents a violation of an action's atomicity. The checking out of a resource is, by our definition, an irreversible action. If an action holding a resource aborts, we must use forward recovery techniques to ensure the resource is returned to the pool or disposed of in some other reasonable way.

Example 1: 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 2: 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. This example is a variation of the resource user presented in Example 1. A variation not shown would involve propogating the capability to the resource into the parent environment so that it could be used by the parent.

```
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
```

Example 3: 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 4: 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 5: 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 runtime 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.[1] This initiates forward recovery within 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
```

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

```
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 6: Some resources may be used in ways which irreversibly change their state, e.g., a missle may be launched. We must insure that such irreversible state changes are adequately recorded, even if the action which caused the change is aborted. In this example, the action becomes irreversible in stage 3. If the action aborts in stages 1 or 2, its effects are undone. If the action aborts in stages 4 or 5, data area 2 is made consistent and permanent. Action 1 then aborts, and an exception is raised. Action 1's parent is responsible for using the information in dataArea 2 when responding to the irreversible launching of the missle. If the action aborts in stage 3, the recovery strategy will depend on whether the launch has been completed.

The when construct reinvokes the action it returns true, raises an exception, or aborts.

The example also illustrates the use of diagnostic operations during recovery. Some of the operations used during recovery will undo effects of operations carried out during the main line of the action. The "undo" operations will have no effect if they are invoked to undo an operation which was never performed. We find that the ability to invoke "undo" operations without knowing whether there is really something to undo simplifies the structure of the abort handler.

```
! dataAreas are global to Action 1
    begin dataArea 1
      missleTracking, targetServer: capability  !initialized elsewhere
    end dataArea 1
    begin dataArea 2
       missle: capability := null
       targetData: a record of some sort
       missleState: (none,available,ready,aimed,inFlight, armed) := none
       outCome: (undefined, notAtTarget, targetHit, targetNotHit, MissleLost) :=
undefined
     end dataArea 2

     begin Action 1 (potentially irreversible)
       stage 1: missle := magazine@obtainMissle()
                missleState := available
                if not missle@systemsOk() then
                  abort
                else
                  missleState := ready
                endif

       stage 2:  targetData := targetServer@obtainData()
                 missle@aim(targetData)
                 missleTracking@noticeOfIntentToLaunch(missle,targetData)
                 missleState := aimed

       stage 3:  potentiallyIrreversible(missle@launch())
2
                 missleTracking@noticeOfLaunch(missle)
                 missleState := inFlight

       stage 4:  when missleTracking@inPosition(missle)
```

[2]the notation "potentially Irreversible" has no effect on execution. Its use provides checkable redundancy with respect to the way the lauch entry point in the missle object is declared

```
                missle@arm()
                missleState := armed


    stage 5: when missleTracking@atTarget(missle)
              missle@detonate()
              outCome := missleTracking@result(missle)

  on exception
    abort

  on abort
    stage 1,
    stage 2:  if failedNestedAction(missle) then
                magazine@possibleMissleFailure(missle)
                roll back data area 2
                raise exception(missleFailure)
              else if failedNestedAction(magazine) then
                magazine@parent@possibleServiceFailure(magazine)
                missle@standDown()
                missleTracking@standDown(missle)
                magazine@parent@return(missle,magazine)
                roll back data area 2
                raise exception(serverFailure)
              else if failedNestedAction(missleTracking) then
                magazine@parent@possibleServiceFailure(missleTracking)
                missle@standDown()
                missleTracking@parent@standDown(missle)
                magazine@parent@return(missle,magazine)
                roll back data area 2
                raise exception(serverFailure)
              else if failedNestedAction(targetServer) then
                targetServer@parent@possibleServiceFailure(targetServer)
                missle@standDown()
                missleTracking@standDown(missle)
                magazine@return(missle)
                roll back data area 2
                raise exception(serverFailure)
              else if internalAbort(self) then
                missle@standDown()
                missleTracking@standDown(missle)
                magazine@return(missle)
                roll back data area 2
                raise exception (internalError)
              else if externalAbort(self) then
                missle@standDown()
                missleTracking@standDown(missle)
                magazine@return(missle)
                roll back data area 2
                restart stage 1
              end endif
```

```
        stage 3: if failedNestedAction(missle) then
                   if irreversible(missle,launch) then
                     missleState := launched
                     raise execption(launchedAndAborted)
                   else
                     missle@standDown()
                     missleState := available
                     raise exception(launchFailure)
                   end if
                 else if failedNestedAction(missleTracking) then
                   missleTracking@parent@serviceFailure(missleTracking)
                   raise exception(launchedAndAborted)
                 end if
        stage 4,
        stage 5:  missleState := missle@statusCheck()
                  raise exception(launchedAndAborted)
    end action
```

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: October, 1987

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Task 1 (Programming Techniques for Resilience and Availability):

Work has continued on a draft of the guidebook to be delivered as a result of this task.

Task 2 (Action-Based Programming for Embedded Systems):

Work is in progress on an extensive distributed calendar example intended to illustrate use of the concepts we have ben developing in a larger context than our previous examples.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

Graduate student Glenn Benson has been added to the project staff and is working on the example under Task 2.

4. Summary of Trips and Meetings

Richard LeBlanc attended the Second IEEE Workshop on Large Grained Parallelism in Pittsburg, PA. Tom Wilkes formerly of Georgia Tech and this project was also in attendance. The 50 researchers invited to this workshop spent two days discussing research in progress in distributed languages and environments, scheduling for parallel programs, real-time models, and operating systems support.

5. Problems or Areas of Concern

No problems or areas of concern are evident at the current time.

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is sufficient to meet the goals of the contract.

7. Related Accomplishments

None.

8. Plans for Next Period

For Task 1, we will continue our work on the handbook. For Task 2, we plan to complete the distributed calendar example.

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	35	1075.5
Research Scientist II	130.5	1691.5
Grad. Research Asst.	261	5190
Secretary/Clerical	87	1566

G-36-645

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: November, 1987

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. Progress

Task 1 (Programming Techniques for Resilience and Availability):

Work has continued on a draft of the guidebook to be delivered as a result of this task.

Task 2 (Action-Based Programming for Embedded Systems):

We have completed the calendar example. The pseudocode is attached to this report.

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 suer 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 illustrated two irreversable actions. An irreversable action is an action that cannot be rolled back after the action initiated. The two irreversable 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 irreversable because multicast may only be implemented by a set of point to point send operation succeeds The multicast operation succeeds if and only if every point-to-point send operation succeeds. The action is irreversable because each atomic send operation is irreversable. The second irreversable 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 irreversable because each multicast is an irreversable 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 irreversable 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 check-pointed list of destinations indicates the destinations to which a precommit message was sent.

The consensus action succeeds if and only if all machines reach a consensus. 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.

In the example there is a situation in which it is convenient for an action to spawn several parallel threads of execution (coroutines). The block of code defining a set of coroutines is delimited by PARBEGIN and PAREND statements.

The use of coroutines has not previously been considered within the Cloud's model. We believe the fact that an action has spawned several coroutines should not evidenced outside the scope of the action. The coroutines may interact through shared objects or shared data areas. It is the responsibility of the object or data area to provide proper concurrency control. The coroutines should not, in general, be treated as actions since they may interact in ways which are nonserializable. Each coroutine may, however, be partitioned into

sequences of nested actions. We will require that it be possible to construct a serializable schedule which preserves the semantics of the coroutines. A rendezvous mechanism will be assumed so that the programmer may place constraints on the scheduling of the actions.

The use of coroutines raises two important issues with respect to fault tolerant computing. First, we must consider means for propagating abort signals in the presence of coroutines. Second, we can consider how to generalize our understanding about how to abort and recover an action in the presence of coroutines to situations involving multiple, independent threads of execution.

We have identified several patterns for propagating the abort signal. For example, support several coroutines have interacted by means of shared objects. If an action within one coroutine aborts and performs forward recovery, then actions within the other coroutines may be required to abort and recover as well. These other actions may be aborted because they have touched objects which were later recovered. In this case the abort signal propagates from the bottom up. We could also arrange for the abort signal to propagate from the original site up to an ancestor where recovery of other actions affected by the abort can be initiated.

The different patterns for propagating the abort signal will give us some additional flexibility when dealing with irreversible and potentially irreversible actions.

Irreversible actions can arise when atomicity is violated and independent threads of execution interact. We believe it is possible to model such interdependencies by regarding the interacting actions as coroutines nested within a common parent. Except for the possibility that an action may end up with multiple parents, this is anologous to the situation involving explicit coroutines. Thus, we believe we can use similar mechanisms for propagating the abort signal. There will be some additional problems related to the coordination of the abort handlers, but we believe these can be minimized by restricting the semantics of the operations which the recovery handlers may perform. Refer to Appendix B for more detail.

2. Special Programs Developed and/or Equipment Purchased

None.

3. Key Personnel

No changes.

4. Summary of Trips and Meetings

None.

5. Problems or Areas of Concern

No problems or areas of concern are evident at the current time.

6. Sufficiency of Effort Toward Meeting Goals of the Contract

The current level of effort is sufficient to meet the goals of the contract.

7. Related Accomplishments

None.

8. Plans for Next Period

For Task 1, we will continue our work on the handbook. For Task 2, effort will be focussed on beginning work on the handbook..

9. Expenditure of Effort

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	35	1110.5
Research Scientist II	130.5	1822
Grad. Research Asst.	348	5538
Secretary/Clerical	87	1653

1 calendar

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
```

1.1 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.

1.1.1 stage 1

Interact with the user through the user interface to obtain a calendar entry. Read the old entry from the calendar (section 1.13) into a log record data structure, and commit the old calendar entry to the log. Any exception raised by the commit action (section 3) is a fatal error. The log entry has the type "precommit". Any query (section 1.12) 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 2.2). *Recover_precom* operates by broadcasting an abort message.

1.1.2 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 1.2) which sends an abort message. If stage 2 procedes normally, all receiving nodes execute the *rec_insert_precommit* (section 1.6) procedure. Otherwise, if an abort message is sent, all receiving nodes execute the *ab_ins_pre* procedure (section 1.8).

1.1.3 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 2.3). *Recover_ins_com* operates by rolling the action forward.

1.1.4 stage 4

Broadcast an "insert commit" message. Any exception is considered fatal. All receiving nodes execute the *rec_insert_commit* (section 1.7). 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 2.6).

1.1.5 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.

1.1.6 stage 6

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

1.1.7 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
            logOclear(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
```

1.2 no_precommit

no_precommit is called by the stage 2 exception handler of *insert* (section 1.1). 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 1.8) when the abort message is received.

1.2.1 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
```

1.3 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 2.5).

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.

1.3.1 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
```

1.4 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 2.6) procedure.

1.4.1 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
```

1.5 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.

1.5.1 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
```

1.6 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.

1.6.1 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
```

1.7 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 2.4).

1.7.1 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
```

1.8 ab_ins_pre

ab_ins_pre is invoked when an abort precommit message is received (see section 1.2). 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.

1.8.1 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
```

1.9 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 1.3) which is called by the exception handler of stage 3 of the exception handler of *insert* (section 1.1).

1.9.1 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
```

1.10 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 2.6) 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 1.4) whenever the *insert* procedure is unable to guarantee consensus among the nodes of a committed entry.

1.10.1 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
```

1.11 queryIO

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

1.11.1 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
```

1.12 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 1.1.1 and 1.1.3), then *query* returns the value stored in the log, otherwise, *query* returns the value stored in the calendar.

1.12.1 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
```

1.13 others

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

1.13.1 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.
!
```

2 recovery object

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

2.1 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.

2.1.1 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
```

2.2 recover_precom

recover_precom is called when a node reaches stage 2 of *insert* (see section 1.1.2), 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 1.9) when the abort message is received.

2.2.1 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
```

2.3 recover_ins_com

Recover_ins_com is called if the insert procedure reaches stage 4 (see section 1.1.4) and then crashes. This procedure implements forward recovery by broadcasting a commit message.

2.3.1 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
```

2.4 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.

2.5 abort_pre

abort_pre is invoked if the node crashes in the exception handlers of either stage 2 stage 3 of *insert* (see sections 1.1.2 and 1.1.3). The exception handlers call *no_precommit* (section 1.2) and *no_agreement* (section 1.3). *Abort_pre* is invoked only if the node crashes in *no_precommit* or *no_agreement*.

2.5.1 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
```

2.6 recov_badcommit

Recov_badcommit is called if a "bad commit" message was placed in the log by *no_commit* (section 1.4). This message indicates a commit is unsuccessful even though all nodes agreed to commit the message. The *recov_badcommit* procedure retries the commit.

2.6.1 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
```

3 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

3..2 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
```

4 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
```

5 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;
```

G-36-645

MONTHLY REPORT

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTING SYSTEMS

REPORTING PERIOD: December, 1987

SUPPORTED BY

ROME AIR DEVELOPMENT CENTER (RADC)

CONTRACT NUMBER F30602-86-C-0032

GIT PROJECT: G-36-645

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

1. *Progress* **Work on both tasks is focussed on the guidebooks that are the primary deliverables of this project. A preliminary outline of the Task 2 guidebook is attached.**

2. *Special Programs Developed and/or Equipment Purchased*

None.

3. *Key Personnel*

No changes.

4. *Summary of Trips and Meetings*

Richard LeBlanc and Win Strickland attended the COTD Technology Exchange.

5. *Problems or Areas of Concern*

Progress was limited this month by substantial leave time taken by project personnel.

6. *Sufficiency of Effort Toward Meeting Goals of the Contract*

The current level of effort is sufficient to meet the goals of the contract.

7. *Related Accomplishments*

None.

8. *Plans for Next Period*

Work will continue on the guidebooks.

9. *Expenditure of Effort*

CATEGORY	HOURS EXPENDED IN THIS REPORTING PERIOD	CUMULATIVE TOTAL OF EXPENDED HOURS
Associate Professor	35	1145.5
Research Scientist II	130.5	1952.5
Grad. Research Asst.	174	5712
Secretary/Clerical	87	1740

Outline for Action Based Programming for Embedded Systems

I. System level fault tolerance: Introduction
   A. Faults, Errors, and Failure: some definitions
   B. Overview of the special requirements of embedded systems

Outline for Action Based Programming for Embedded Systems

I. System level fault tolerance: Introduction
   A. Faults, Errors, and Failure: some definitions
   B. Overview of the special requirements of embedded systems
   B1/2. Design Process
      1. Architectural model
      2. Specification
      3. Implementation
   C. Regard language used in examples as a design language. Since compilers not widely available, constructs must be translated by hand.
   D. Importance of layered structure
   E. Designing lower layers to provide services to support fault tolerance
   F. Using the Handbook
   G. Types of Faults
      1. Transient faults
      2. Temporary faults
      3. Permanent faults
   H. The sources and inevitability of faults
   I. The limits of fault tolerance mechanisms
   J. Other uses for recovery mechanisms
      1. Simplfy systems management
      2. Change status of site without detailed coordination at other sites
      3. Software and Hardware maintenance and upgrades without interrupting system services
   K. The special role of redundancy in acheiving fault tolerance
      1. Try the computation again
      2. wait and try the computation again latter
      3. Try an alternative computation
         a. vary the code
         b. vary the hardware or site
         c. vary the data
   L. Degrees of Fault tolerance
      1. What is the range of possibilities
      2. The degree of fault tolerance which is acceptable is requirements driven
   M. Aspects of Fault tolerance
      1. availability
      2. resiliency
      3. forward progress
      4. reliability
      5. there are trade offs among these aspects of fault tolerance
      6. the relative importance of these aspects of fault tolerance is requirements driven

II. Requirements
   A. Atomicity

1. size of atomicity
2. criticality

B. **Functional characterization**
   1. **Time preservation**
   2. **Frequency**
   3. Criticality

C. Synchronization
   1. Criticality
   2. **Number of functions**
   3. **Interface with architecture**

D. Time
   1. Normal Operations
   2. Switchover Time

E. Processing Requirements
   1. Operating System
   2. ALU
   3. Mass Memory Organization

III. System Architecture (Design choices supporting Faul tolerant system design)
   A. Switchover Characteristics
      1. Location
      2. Processor State
      3. Processor Load
   B. Redundancy Approach
      1. embedded
      2. Dedicated
      3. shared
      4. distributed
   C. Processor Characteristics
      1. memory size
      2. mass storage size
      3. processor speed
      4. bus architecture
      5. other

IV. Selection of Redundancy Approach Function (the choice of architecture is requirements driven)
   A. Atomicity
   B. Functional Characterization
   C. Synchronization
   D. Processing requrirements

V. Software Architectures
   A. The choice of Software Architecture is driven both by requirements and System Architecture
   B. The Basic Software Models
      1. Process Models (message passing)
      2. Object/Action Model
         a. actions for concurrency
         b. actions for fault tolerance
         c. nested actions
         d. actions which terminate before they commit
         e. actions which terminate after they commit (irreversible actions)
            i. explicit commit

ii. implicit commit
1). communication with actions in other scopes
2). interaction with the external environement
f. irreversible actions have the effect of weakening failure and concurrency atomicity
3. Hybrid models (needed for embedded systems)
a. interleaving actions embedded within processes
b. constraining how actions may be interleaved
c. the problem of message passing in an enviroment structured in terms of nested actions
d. interleaving the actions nested within other "weakened" actions
e. "weakened" atomicity must be accompanied by strengthened requirements for consistency

VI. Choices in the design of fault tolerance mechanisms
A. choices are driven by requirements, system architecture, and software model
B. The basic mechanisms
1. Backward recovery
2. Forward recovery
3. How context sensitive should the semantics of recovery be?
C. Constructing a consistent state
1. in anticipation of failure
a. logging
b. shadows
c. replicating the current copy
2. after failure
a. deriving values from other data
b. deriving values by merging replicas
c. deriving values from the environment
d. deriving values from archival data
3. mixing strategies
D. Dealing with data missing following recovery
1. how data related to the occurence of an irreversible operation may be lost
2. after recovery (data of low criticality may not be available until after computation has resumed)
3. Re-execute the operation
4. Inquire as to the state of other physical or data objects to determine whether the operation occured
5. Specialized hardware to ensure the event is logged and not lost
6. cutting over to reduced level of service
a. permanent cut over
b. temporary cut over
E. Dealing with hardware failures
1. backup hardware available
2. partial failure of hardware, no backup
3. failure correctable with software intervention
4. failure not correctable except with human intervention
5. returning hardware to service following repair
F. Dealing with processes which have read data invalidated by recovery
1. finding them
2. deciding which should live and which should die

3. coordinating recovery among independent threads of control

G. Resuming computation

1. on the same machine
2. on a different machine
   a. protocols for triggering resumption on a backup machine
   b. protocols for triggering resumption on a primary machine
      1. when there is a designated primary machine
      2. when several machines compete for backup responsibility

H. Selecting the code to be run after recovery

1. retry the code which failed
2. execute alternate code
3. resume at some point subsequent to the failed code
4. other possibilities
5. making the new code a permanent replacement for the failed code

I. responsibilities of the recovery handler and other code

1. how much should be done during recovery by the recovery handler and how much by the resumed comutation?
2. does the code in which computation resumes need an "on recovery" handler as a preamble?
3. should recovery be bound to the actions which failed or to the objects which must be recovered?
4. does forward recovery need both exception handlers and recovery handlers?

J. Recovery in nested actions

1. propogating recovery over several levels of nesting
2. some guidelines for designing multilevel recovery

K. Dealing with orphans

L. Coordinating the design of an actions forward recovery and the design of objects

M. Faults in the recovery handlers

VII. Implementing these ideas

A. providing system services

B. translating constructs into a conventional programming language

Fault Tolerant Software Technology for Distributed Computer Systems

Interim Technical Report

February 18, 1987

F30602-86-C-0032
G36-645

Richard J. leBLanc
C. Thomas Wilkes
Stephen Orburn

1. Introduction

This report documents the research results of the project entitled "Fault Tolerant Software Technology for Distributed Computing Systems" during the first year of its two year term. The report is divided into two major sections, corresponding to the two subtasks of the project. The first of these summarizes the research which has been performed for the task "Programming Techniques for Resilience and Availability." Separate subsections are devoted to each of the major facets of this research, including:

- consideration of problems of replication endemic to object-oriented systems with general (i.e., non-flat) object structure;
- investigation of naming schemes for support of replication which deal with these problems in object-oriented systems;
- consideration of the relationships between our research on replication of objects and recent research by others on replication of abstract data types;
- investigation of the role of the fault-tolerant job scheduling system in dealing with the problems descibed above; and
- investigation of the design of the Object Filing System as a possible source of a paradigm for replication in action/object systems such as Clouds.

Finally, this section concludes with a summary of the status of this research and the results to date, and a comparison of our research to the directions taken by the similar project on the topic "Fault Tolerant Distributed Systems" at Honeywell, Inc.[Hone86]

The second major section describes our work on the task entitled "Action-Based Programming for Embedded Systems." Since our work on this task has only been in progress for about seven months, this section is more of a discussion of the issues that have been identified than it is a presentation of results. The major issue is the seeming incompatibility of the idea of atomic actions with the irreversible operations frequently preformed by the software of embedded systems. We consider the problem of preserving information about irreversible operations performed by an action when an action aborts, possible mechanisms for performing this task, and problems related to classifying various types of irreversible operations. Finally, plans for further work on this task are presented.

2. Programming Techniques for Resilience and Availability

The research reported in this section has been significantly influenced by a variety of related efforts that are part of the Clouds project. Our discussion frequently references Clouds, but it should be recognized that this work is relevant to any object-based system that supports some form of atomic actions.

2.1 Problems of Replication in Object-Based Systems

In the course of our 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).

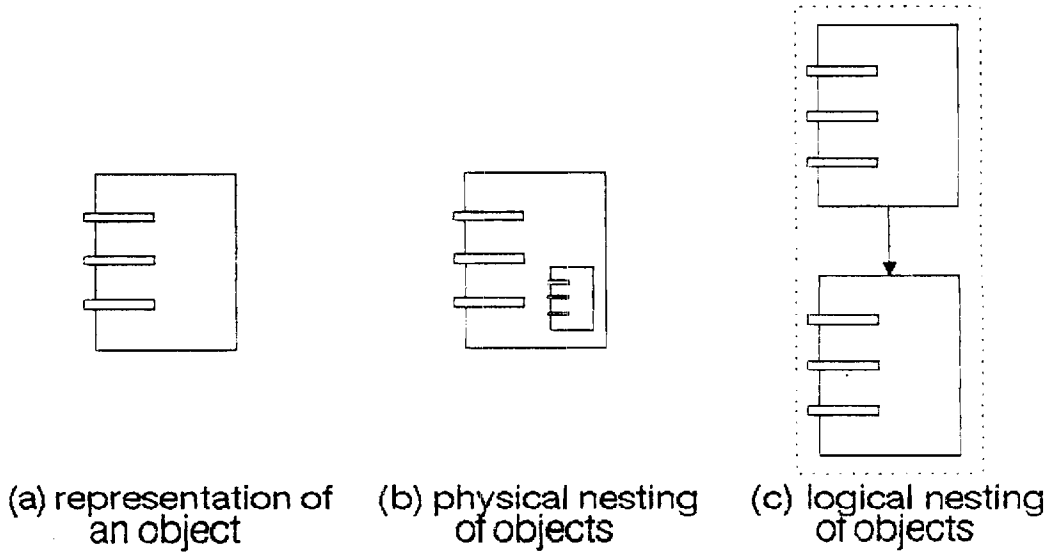


Figure 1. Graphical Representation of Object Nesting

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 and from registering the capability with the 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*. (A graphical 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 the OFS, we say that *B* is 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 the computation from which the desired state results on each replica; we refer to this scheme as *idemexecution*. Another method is to execute the computation at one replica, and then copy the state of that replica to the other replicas; we refer to this scheme as *cloning*. (Representations of the idemexecution

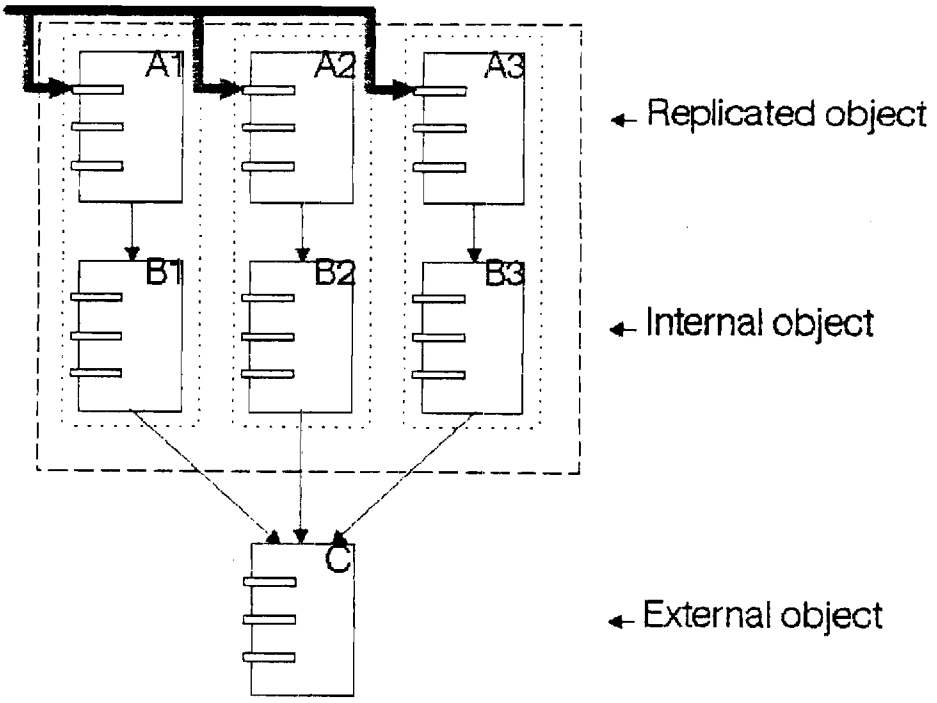


Figure 2. Replicated Object with Internal and External Object References

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 our 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 other replicas is lost. (This problem is represented in Figure 4.)

Our current research includes an investigation of a "taxonomy" of object structures on which the corresponding state-propagation methods may be safely used, as well as of how these state-propagation methods—or the Clouds object-

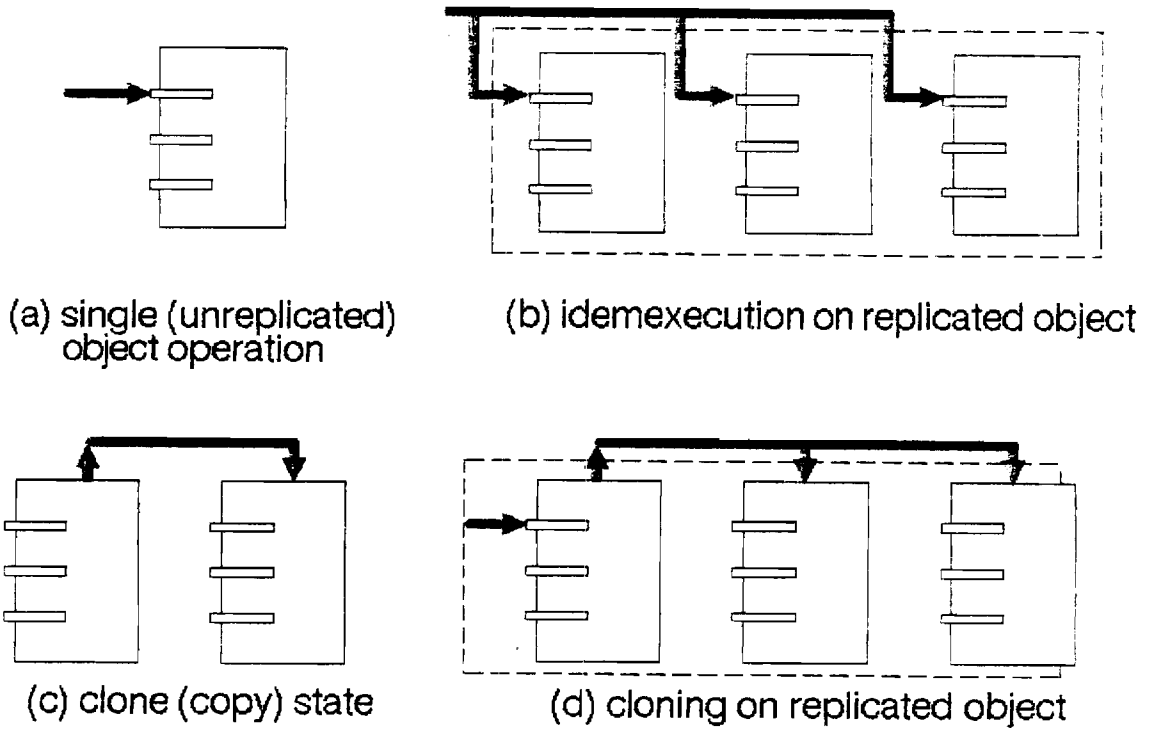


Figure 3. Replicated State-Copying Methods

naming mechanism—may be altered to safely handle more general cases. Our current feeling is that the latter may be achieved with minimal alterations to the kernel, via having the kernel interact with the Object Filing System and the fault-tolerant job scheduler, two services of the Clouds system which are described in more detail in the following sections. We also discuss some schemes developed during the preceding year for replicating actions and for naming of replicated objects, which should aid in achieving these goals.

2.2 Naming Schemes to Support Replication

We are currently considering two different capability-based naming schemes which may be used in support of *state cloning*, as described in the previous section. The first scheme requires minimal changes to the 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. (We do not consider this constraint 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. This constraint may be enforced by the Object Filing System, which is described in a later section.) 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.

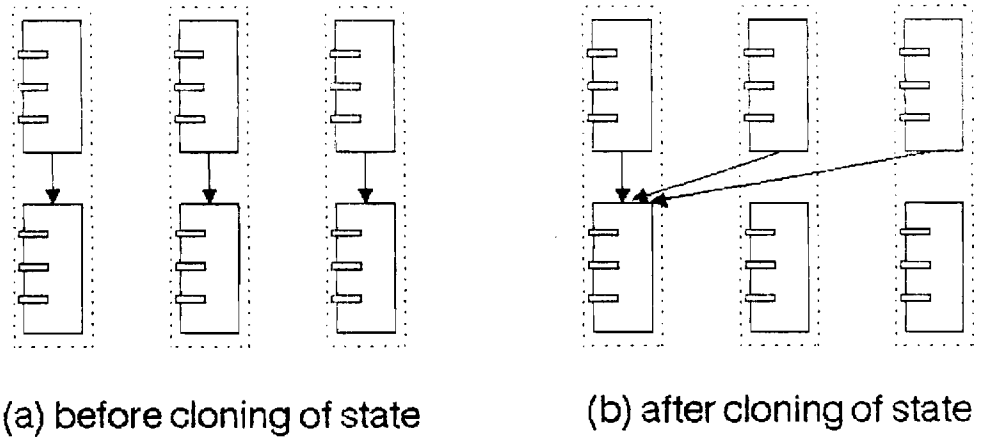


Figure 4. State Cloning with Internal Objects

Although we will thus have 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 mentioned in the previous report (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. We call this scheme *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 from, for instance, 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, we refer to the scheme as *horizontal replication*. One such naming scheme is described in a paper which we have co-authored with other Clouds researchers,[Aham87] which is included as an appendix to this report.

We are currently considering the merits of each of these naming schemes in the context of the *replicated actions* scheme, which is described later in this report.

2.3 Related Research on Replication of Abstract Data Types

As part of our work on achieving availability of resources in the Clouds system, we have been involved with study of the work of Herlihy, presented in his dissertation, "Replication Methods for Abstract Data Types,"[Her184] and with correspondences between Herlihy's techniques and the synchronization mechanisms used in Clouds, which should allow us to apply Herlihy's methods to our problem of generating replicated objects.

Herlihy's work 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.

There appears to be a close relationship between Herlihy's quorum intersection graphs and the lock compatibility matrices used in Aeolus and the Clouds system; a graph constructed from the lock compatibility matrices for an Aeolus/Clouds object is either the complement of the quorum intersection graph for the operations of that object, or a subset of the complement. This is not really surprising, since the specification of our lock compatibilities is based on the programmer's analysis of the compatibilities between the object operations, while Herlihy's quorum intersection graph may be viewed as being based on an analysis of the *incompatibilities* between operations.

Thus, we should be able to apply Herlihy's techniques to our problem of generating replicated objects given an unreplicated object version and a specification

of the desired replication properties. This entails extending the notion of the Aeolus/Clouds lock to include the "distributed" lock; that is, the state of the lock is shared logically among all replicas of an object. This will, of course, require the transmission of lock state information among all replicas. However, the concurrency properties of the unreplicated version of the object will be retained by the replicated version generated from it. This is especially significant given the power of the Aeolus/Clouds lock mechanism in expressing arbitrary compatibilities and in allowing the expression of synchronization at arbitrary levels of granularity.

Another interesting question—which is being be investigated using the Object Filing System example (described in a later section)—is the relation between Herlihy's quorum intersection graphs and Aeolus/Clouds lock compatibility matrices when multiple locks are used for specifying an object's synchronization behavior. We have found that in certain cases it is convenient to use more than one lock to specify synchronization among an object's operations; the different locks typically apply at differing levels of granularity as well as having compatibility matrices with disjoint meanings. For example, we have designed a symbol table object which uses two locks for synchronization purposes: one lock at the level of the individual buckets in the symbol hash table, with compatibilities expressing a multiple reader / single writer protocol; and another lock at the level of the entire symbol table, allowing multiple readers or multiple writers, but not readers concurrently with writers. The first lock is used with the typical operations such as insert, delete, and find, where there is no interaction between concurrent operations on different buckets; the second lock is used with an "exact-list" operation, where a "snapshot" of the exact state of the symbol table at a particular instant is desired, and thus all operations which modify the state of any portion of the symbol table must be locked out. (This symbol table object is described in more detail in the section on the Object Filing System, and the Aeolus code for this object is included in an appendix.) Our locks thus have an advantage of power of expression over Herlihy's quorum intersection graphs, which do not allow the expression of granularity lower than that of an entire abstract data type and its operations. Thus, we are considering how Herlihy's results may be extended to the case of multiple levels of granularity for synchronization.

2.4 The Fault-Tolerant Job Scheduling System

Our current work on the study of programming methodologies appropriate to distributed systems involves the study of various methods of achieving resilient, available objects through the use of replication. Similar work[Birm85, Birm85a] has recently been reported by researchers on the ISIS system at Cornell; however, that work (unlike ours) does not consider the problems introduced by network partitions, assuming rather that all failures are of the so-called *fail-stop* variety. In our work, we take into account the problems involved in reconciling the states of replicated objects which have run in independent partitions during a network failure. Thus, we

may achieve higher availability in situations in which temporary violations to consistency are tolerable. Our work, as well as recent work[Dasg86] by other researchers in the Clouds project, has also suggested some of the functionality which will be required of the fault-tolerant job scheduler for the support of availability in Clouds. It is in the job scheduler that we envision most, if not all, of the knowledge about individual machines in the system will be concentrated, such as whether a certain machine is available or what the current loads are on the individual machines. Thus, the job scheduler is the natural portion of the system to support functionality such as the creation of distributed replicas of an object class, the selection of the most appropriate individual replica from a class of such replicants to perform work requested of the class, or the support of *forward progress* (that is, moving work started on an object running on a system which subsequently failed to another system on which another replicant of the object exists). We anticipate that our work will provide a firmer design for the interface needed with the job scheduler.

2.5 The Object Filing System

We are currently investigating methodologies for resilience in action/object systems in the course of the design of the object filing system (OFS) for Clouds. The replication scheme which we are currently considering in support of availability will require heavy interaction between the manager for a replicated object, the job scheduler, and the OFS. The OFS should:

- be resilient and highly available (through replication);
- provide a mapping from object names (strings) to Clouds object capabilities;
- impose some familiar structure (e.g., a Unix-like hierarchical structure) on the flat, global system name space provided by the Clouds object manager;
- provide efficient forms for the most common types of I/O (such as text I/O) without the necessity of the context switches which would be required if such I/O were modelled with Clouds objects.

In the OFS, an object name may represent a *group* of objects (the set of replicas of a replicated object), rather than a single instance. We intend that this mechanism should be, in general, transparent to the user (although special-purpose applications—such as DBMSs—may require that, in addition, finer control of replication be available than that provided by a general mechanism).

To test this conjecture as well as the possibility of deriving replicated objects from single-copy versions, we have been occupied during the previous few months with the specification and implementation of a single-copy version of a prototype Object Filing System (OFS) for Clouds. (A much-simplified version of an OFS—called the NameServer—supporting a flat name space has already been implemented in the Clouds system, while the current OFS effort will support a hierarchical name

space.) This effort involves the specification of the synchronization behavior of the single-copy OFS object via a lock compatibility matrix. We will compare the graph derived from this specification with the quorum-intersection graph appropriate to the same set of operations. Also, now that the specification of the single-copy OFS object is complete, we are testing our idea of extending the single-copy version to a replicated version by allowing the locks specified for the single-copy version to act as "distributed locks" (where information about locks granted or released by a replica on one node is communicated to the other nodes where replicas exist by the Clouds object manager). A "distributed lock" may be viewed as a manager for gathering a quorum for a given operation; the synchronization behavior thus achieved should reflect that specified for the single-copy version, with no additional effort on the programmer's part.

2.5.1 Overview of the OFS Design The prototype OFS design involves a hierarchical nesting of "OFS" objects, each of which maintains knowledge of its immediate ancestor in the hierarchy. (Here, by "nesting" we mean "logical nesting," as described above.) The children of an OFS are stored in and accessed through a symbol table object nested in the OFS. The design of the symbol table object has been described in one of our publications;[LeBl85] the locking structure of the symbol table object was described briefly in our last report. (We have already investigated the recoverability properties of the symbol table object; the recoverable version of the symbol table is described in the publication mentioned above. Also, we have developed a replicated version of the symbol table object using techniques somewhat different from those we are using on the OFS object. However, we will be ultimately using the recoverable, non-replicated version of the symbol table object in the OFS, leaving the implementation of replication at the level of the OFS object.)

This logical structure of the OFS object is represented graphically in Figure 5 (a). Here, the internal structure of the OFS object is shown, with its nested symtab object, which may have capabilities (represented by arrows) to other OFS objects as well as non-OFS objects. In Figure 5 (b), an example hierarchy constructed with OFS objects is shown. Here, an OFS object is represented by a single object (hiding its internal structure). The OFS objects in the hierarchy are shown as icons with no pattern fill, while non-OFS objects are filled with a pattern.

The interpretation of user commands to the OFS is handled by a rudimentary "shell" process, which accepts Unix-like pathnames and translates them to operations on an OFS or invocations of Aeolus processes. The "shell" process maintains knowledge of the root of the OFS hierarchy, as well as the "current" OFS (corresponding to the "current working directory" in Unix), and (for efficiency purposes) the ancestor of the "current" OFS.

The synchronization mechanism already in place in the nested symbol table objects suffices for the synchronization of the OFS objects as well. This simplifies the analysis of the compatibility matrix/quorum intersection graph relationships,

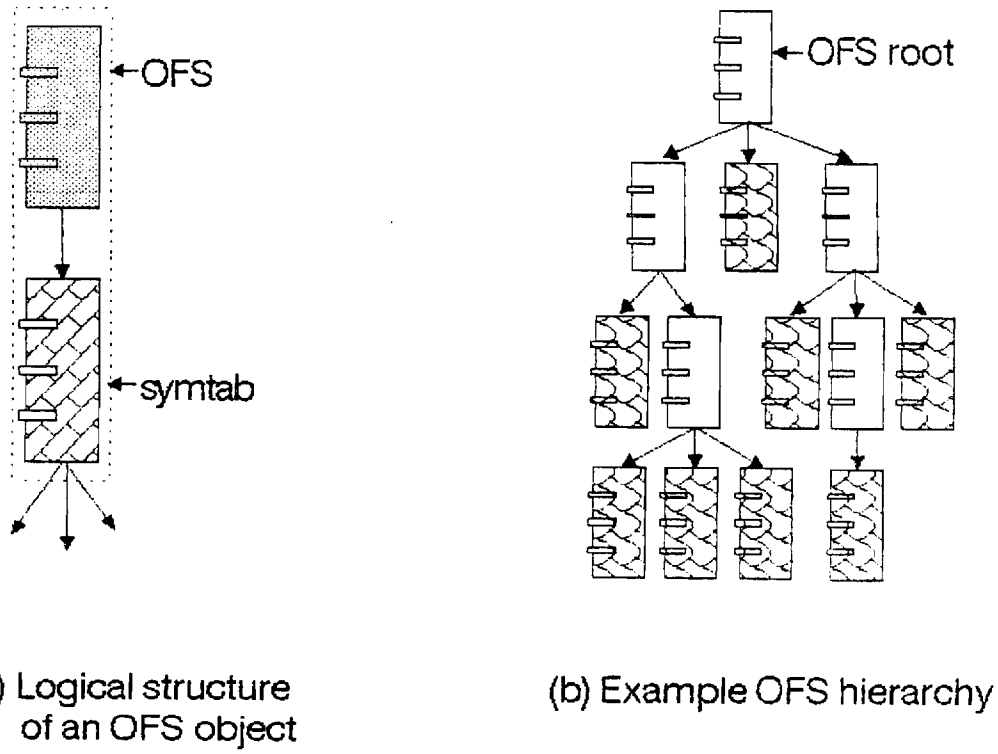


Figure 5. OFS Structure

since the symbol table object has fewer operations than the OFS object (five as opposed to eleven[1] operations), and thus fewer interactions among the object's operations, yielding simpler compatibility matrices. Since the symbol table object synchronization involves two locks at differing levels of granularity, we are investigating how such locks relate to quorum intersection graphs when the latter are extended to multiple levels of granularity.

2.5.2 Detailed Discussion of the OFS Object The Aeolus source code for the Object Filing System and supporting objects is given in Appendix B. For each object, both a definition part (with extension ".def") and an implementation part (with extension ".imp") appear; the source code for an Aeolus process is given the extension ".pro". These objects include the OFS object itself (**OFS**), the symbol table object mentioned above (**symtab**), the shell-like driver process (**shell**), and a supporting object which performs parsing of command lines (**Names**).

1. The original interface of the OFS object had eleven separate operations. The OFS definition recently has been redesigned to consolidate six of these operations having a common interface into a single operation, called **general_op**; this definition is included in Appendix B. However, if locking were taking place in the OFS object, each different function of **general_op** might require a separate locking mode. Thus, the effect would still be of eleven separate operations on the OFS.

All of the Aeolus objects in the versions mentioned above are defined to be *local objects*, that is, objects which do not use the Clouds object management facilities, but rather are supported by the Aeolus runtime system alone. A major difference between local objects and true Clouds objects is that each Clouds object exists in its own virtual address space, while a local object exists within the address space of the object or process which instantiates it. (Thus, a local object may be regarded as being *physically nested* within its instantiator, as well as being *logically nested*, as are Clouds objects.) As a result, although a local object may not be accessed outside of its instantiator, nor persist beyond the instantiator's lifetime (as may a Clouds object), there may be less overhead involved in access to a local object than there is with a Clouds object, which involves interaction with the kernel's object management system and the mapping of the invoked object's virtual address space.

Although these objects are shown in their local object versions, the Aeolus language allows the programmer to convert such objects to true Clouds objects with relative ease. All that is involved to convert the objects shown to yield (non-resilient) Clouds objects is to change the keywords **local object** to **nonrecoverable object** in the definition parts of the appropriate objects. Note that no changes to the implementation parts of these objects are required.

It is sometimes more difficult to change a non-Clouds object or a non-resilient Clouds object to a resilient Clouds object. Again, this change is possible with the change of a single keyword, in this case from **nonrecoverable object** to **autorecoverable object**. However, an *autorecoverable* object may often be inefficient, since this involves making the entire persistent state of the object recoverable, and thus each action touching such an object will get a new version of the complete object state. Attempted use of automatic recovery (as well as recoverable areas in general) may also be unworkable if the object in question keeps some of its persistent state in the non-permanent heap. Placing pointers into the non-permanent heap into a recoverable area does not make the memory which these pointers reference recoverable; the effect would be analagous to attempting to make a book recoverable by making a spare copy of its index: if the book is destroyed, having a backup copy of the index does not suffice to reconstruct the book. What is needed is that the portion of the state which, in non-resilient objects, is kept in the non-permanent heap, must be kept in the *permanent heap* in resilient objects. This requires some recoding of non-resilient object versions, since such objects have no support for the permanent heap; only *recoverable objects* have permanent heap support available.

The OFS object requires little reorganization to be made into a resilient object. Essentially, the only state which requires recovery are the OFS's knowledge of its current pathname and the capability to its nested symtab object. If the symtab object itself is to be made recoverable, on the other hand, extensive rewriting is necessary, since a major portion of its state is kept in the heap. Thus, in Appendix B we also

show a resilient version of the symtab object (called **r_symtab**) which makes extensive use of the *per-action variable* and permanent heap contructs of Aeolus to simulate the effects of recoverable areas at possibly significantly reduced overhead, as well to allow resilient structures to be allocated dynamically.

Since the symtab object nested in an OFS may be of a different object classification than the OFS object in which it is nested, there are several useful combinations of differing OFS and symtab versions.

TABLE 1. Feasible Internal Organizations of OFS

	OFS	**symtab**
	object classification	
non-resilient	local	local
	nonrecoverable	local
	nonrecoverable	nonrecoverable
resilient	recoverable	(permanent) local
	recoverable	recoverable

Those combinations which are considered "reasonable" are shown in Table 1. These combinations are grouped into those which yield non-resilient objects, and those which yield resilient objects. For example, the first combination in this table is of the local object version of OFS with a nested local object version of symtab, yielding a non-resilient OFS object. Some theoretically possible combinations have been omitted from this table as not being useful. For instance, it would be possible to nest a recoverable version of symtab in a nonrecoverable or even local version of OFS, but the result would be non-resilient, since a crash of the OFS object could result in it losing its state, including its capability to the nested symtab object. This combination would thus be wasteful because of the overhead involved in the recoverable version of symtab.

The combinations listed in the table have also been represented graphically. The combination of an OFS object and a symtab object, both of which have been generated as Clouds objects (either nonrecoverable or recoverable), has already been represented in Figure 5 (a). The combination of OFS and symtab where both are non-Clouds (local) objects is represented in Figure 6 (a); here, the local OFS object is shown physically nested within the shell process. The combination of a Clouds OFS object with a nested non-Clouds (local) symtab object is represented in Figure 6 (b); in this case, the shell process contains a capability to the OFS object, which is thus logically nested within the shell process.

Of the non-resilient combinations, perhaps the most interesting are that combining local versions of both OFS and symtab, as well as that combining the

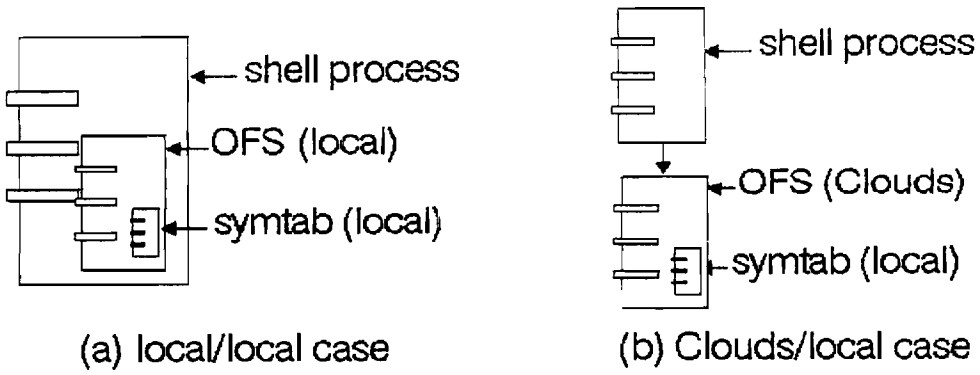


Figure 6. Alternate Internal Organizations of OFS

nonrecoverable version of the OFS with the local version of symtab. At least for testing purposes, the use of a non-Clouds version of OFS is feasible (even though the filing system root would thus be a non-Clouds object) because Aeolus processes actually reside in "lightweight" Clouds objects called **ProcessManagers**. Thus, the shell process for such a combination would maintain the filing system in its persistent state. The combination of nonrecoverable versions of both OFS and symtab would probably have no advantages over the nonrecoverable OFS / local symtab combination; indeed, the overhead of the Clouds system involved in access to the nonrecoverable version of symtab would most likely be a disadvantage for the former combination. However, it should be useful to compare the actual performance of these two combinations to see if this is the case.

In fact, the implementation of hierarchical directory structure using nested nonrecoverable OFS objects requires at least one invocation of a different Clouds object for each nested directory in a pathname. As mentioned above, under the prototype implementation of the Clouds kernel, a Clouds object invocation involves mapping that object's virtual address space into user space. We have little experience yet with the effect of this implementation of object invocation on performance. With the combination of the local versions of the OFS and symtab objects, however, we avoid invocations of Clouds objects during traversal of the directory structure. However, as presently designed, this version of the OFS is not practical except for testing purposes, since each instance of the shell process would get its own copy of the process state (and thus of the complete OFS); thus, changes made in the OFS in one instance of the process would not be available to other instances. (That is, one user would not see changes made by another user.) Removing this problem would require making at least the root of the OFS a Clouds object with a well-known capability, even if the rest of the OFS hierarchy consists of local objects; this dichotomy between the root OFS and the nested OFSs would introduce a new set of problems. However, we feel that comparison of the two OFS combinations will be a valuable study in the methodology of programming under Clouds.

The first of the two resilient combinations in the table bears some explanation. Here, a local version of symtab is nested in a recoverable version of OFS. However, in the recoverable version of OFS, the variable holding the capability to the instance of symtab is declared to be *permanent*; for a local object, this means that the state (data area) of that instance of the local object will be allocated in permanent storage. Invocation of the *modify* operations of such a local object is restricted to toplevel precommit time; however, *examine* operations may be invoked at any time. Such a combination would require some rewriting of the OFS object (not shown here) to avoid violation of this restriction. (The above restriction does not apply to a Clouds object instance, even if the variable holding its capability is declared to be permanent; in any case, such a declaration has no effect on the allocation of the state of a Clouds object, since that state is in a separate virtual address space.)

2.6 Status and Comparison to HOPS

In this section, we provide an overview of the status of our work on programming methodologies, and compare this work (in its context of the Clouds system) to the related project at Honeywell, Inc.

2.6.1 Status of the Research Our understanding of the problems involved in replicating objects of arbitrary structure has increased during the past year, in part due to the research into naming schemes described earlier in this report, and also due to the development of the *replicated action* scheme, which is described below in relation to the work at Honeywell. Given one of the naming schemes which we have developed, we now understand how to use the *cloning* method of propagating state among replicas of an object without losing information about the internal objects of the replicas. The matter of which of these naming schemes is more appropriate for use in the Clouds system is now being considered in the context of the needs of the replicated action scheme.

During the preceding few months, we have completed the design and implementation of both the nonrecoverable and the recoverable versions of the Object Filing System. We have also done extensive work to the Aeolus compiler and runtime system to allow testing of the OFS as a collection of Clouds objects under the control of the Clouds kernel. Testing of the nonrecoverable version of the OFS now awaits only the implementation of Clouds kernel support for locks by members of the kernel team, which should require relatively little effort. Then, we will be able to determine the performance penalty imposed by Clouds objects in this version of the OFS in comparison to the OFS version using (non-Clouds) local objects. Testing of the recoverable version of the OFS awaits the full implementation of action management by the kernel team.

In the meantime, the OFS design is serving as a case study in the methodolgy of replicating both objects and actions in Clouds. We have investigated how Herlihy's

results for determining quorum intersections for the operations of an abstract data type may be applied to the compatibility matrices associated with Aeolus/Clouds locks, as described earlier in this report, and have developed the concept of a *distributed lock* which is conceptually shared among the replicas of an object. The extension of the single-copy version of the OFS to a replicated version is sharpening our intuition as to how the relation between Herlihy's quorum intersection graphs and our compatibility matrices may be extended to multiple locks in the same object. Also, during the preceding year a scheme has been developed for replicating actions to achieve fault tolerance (described below); the OFS is serving as a testbed to test these ideas.

2.6.2 Comparison to 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.

Thus, it is clear that the research involved in the HOPS project closely parallels our research, in the course of which we hope to rectify what we feel is a dearth of experience in programming for fault-tolerance in object/action systems, yielding a framework in which fault-tolerant servers may be constructed in Clouds.

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.

In conjunction with other researchers in the Clouds project, we have been examining a new scheme for fault tolerance. In this scheme, we replicate not only objects (data) but also actions. Each of these *replicated actions* runs as a nested action and has its own thread of execution, each of which is referred to as a *parallel execution thread* (**PET**). An introduction to the PET scheme is given in a paper co-written with the other Clouds researchers,[Aham87] which is attached as an appendix to this report. 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 states of those objects touched by the committed action are then cloned to the other replicas of those objects.

Together with other researchers in the Clouds project, we are currently involved with designing the lower level algorithms and modifying the Clouds action management scheme to implement the PET scheme. At a higher level, we are also considering other implications of the use of replicated actions for providing fault tolerance. We believe that our research on replicated actions complements the DRB / distributed conversation approach taken in HOPS; for instance, it is not difficult to implement the recovery block mechanism in terms of the action facilities provided by Aeolus/Clouds. Comparisons of the results obtained by these two approaches should prove of interest.

3. Action-Based Programming for Embedded Systems

3.1 The Problem

Programmers and programming teams frequently adopt conventions which constrain program structure and which standardize the strategies for coupling components and controlling execution. While such conventions are, in principle, unnecessary, well chosen conventions will help a programmer conceptualize his work. By following agreed upon conventions, furthermore, a programmer will produce a program text which can be read, understood, and checked by his colleagues. A study of the history of programming languages reveals programmers have developed pseudo-code for expressing some of the more useful conventions. In many cases it has been practical to process the pseudo-code algorithmicly; this may

be done by means of a pre-processor which checks the pseudo-code or translates it into a language for which a compiler is available. In a few cases the conventions have proved so useful constructs supporting them have been included in newly designed programming languages—this has been the case with subroutines, user defined data types, software modules, exception handlers, and more recently, with objects and actions. The handbook described herein will consider possible programming conventions, and supporting language constructs, for building fault tolerant software for applications in which some operations may be irreversible.

Large systems, such as the command and control systems required for many military applications, are likely to perform complex missions; the overall system will be multifunction—some functions involving human/computer interaction and others the control of some mechanism or process. Such large systems are likely to be distributed and to incorporate as subsystems 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.

Systems this large cannot be expected to be without flaw. Inevitably there will be *errors* in the requirements analysis, systems architecture, and in the design and coding of components. 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 system functionality or performance as perceived by human users or client systems, a *failure* is said to have occurred. Often considerable time will have elapsed between the fault and a resulting failure. The failure may be the cumulative effect of a number of faults.

In a *fault tolerant* system, it is possible to detect, sooner or later, the occurrence of a fault. On detecting the fault, a fault tolerant system will attempt to intercept the ensuing chain of events by:

1. containing the consequences of the fault and preventing the fault from "infecting" other portions of the software;

2. attempting to "repair" the damage done by the fault before it could be contained. This includes attempts to mitigate the effects of the fault on system function and performance; and

3. addressing the fact that an error has been uncovered in the system. Depending on the nature of the error and the fault it has caused, any of a variety of corrective measures may be deemed appropriate.

The programming techniques, software designs, and operating philosophies which can be used to implement these measures will be described in the handbook. Several are illustrated in the last section of this report.[2]

Many large software systems run on dedicated computers or, as is likely, on a dedicated network of computers. In dedicated systems much of the support normally provided by a general purpose operating system is instead provided by the software system itself. This is often done in order to satisfy various standards related to system performance. A large software system can be thought of as having two layers. An external, applications layer consists of the software which provides services to the system's clients—perhaps human, perhaps not. An internal, systems layer provides the applications layer with an interface to the underlying computational resources and, in doing so, takes on much of the responsibility for managing those resources. The systems layer may provide such services as file handling, telecommunications, access control, and management of such resources as memory, processing units, and peripherals. It can be argued that it is most important to provide fault tolerance in the systems layer because failures there may not only compromise the system's ability to perform certain functions, but may cause the system to shut down entirely.

Fault tolerance is most critical in the internal, systems layer. Since a systems routine may support all of the applications, a failure there can compromise many or all of the system's functions. Faults in the external applications layer will generally compromise one or at most a few functions.

The *Clouds* system represents the results of several investigations into the problems of constructing reliable, distributed systems. Among the goals of the *Clouds* project is that of contributing some fundamental insights into the problem of structuring large, multifunction systems in a distributed computing environment. Of special interest are the problems of establishing fault tolerance in a system which is "open," i.e., programmers may readily enhance, extend, optimize, tailor and otherwise maintain applications and systems services. *Clouds* also allows a substantial amount of data and code to be shared among applications and among service routines. This sharing is accomplished by encapsulating the shared data and code within persistent objects. In contrast to most open systems, *Clouds* is a multi-user/multi-application system, and this presents a number of problems in in extending and maintaining shared objects. Strategies must be developed which allow programmers to perform maintenance without interrupting service to clients and

2. The examples found there emphasize recovery techniques which are appropriate for coping with actions which fault or abort after performing an irreversible operation. The last example illustrates how a programmer may actually simplify the structure of his software by declaring certain operations to be irreversible along with providing appropriate means of recovery.

which prevent errors in one application or systems level service routine from corrupting the data and code used by others. Maintenance within this environment will also present complex problems in version control; in particular, maintenance may result in various system components having different expectations regarding a shared object's interface and semantics. Fault tolerant designs are needed if clients are to be protected from programming errors in general and the problems of inconsistent versions in particular. In the absence of such fault tolerance, errors in the way objects are constructed or interfaced could result in a cascade of failures across the entire system.

For many of the situations which can be expected to arise in a *Clouds* like environment the conventional techniques for achieving fault tolerance, such as roll back/retry, will be inadequate: the operation which faulted may have also caused irreversible changes in the state of certain non-recoverable objects or in the systems being controlled. The final results of the investigation will address the problem of of achieving fault-tolerance in the presence of such irreversible operations.

The techniques related to this task can be divided into three fundamental classes:

1. mechanisms for preserving information regarding irreversible operations which may have been performed within an otherwise atomic action;
2. strategies for using the information so preserved to construct a consistent and correct state from which computation can be resumed; and
3. new programming techniques, system designs, and operating philosophies made possible by the availability of an enriched set of means for achieving fault tolerance.

3.2 Fault Tolerance and Atomic Actions

The *Clouds* system and its programming language *Aeolus* incorporate the concept of *atomic action* in a fundamental way. An atomic action is one which executes completely or not at all; it is not visible in a partially executed state. In incorporating the concept of "irreversible operation" into this model of computation, the concept of an atomic action has required some revision. While the notion of "atomicity" has remained unchanged, that of "complete execution" has been modified. In the original model an action encountering a fault was required to abort and was regarded as having never occurred. This usually entailed rolling the state of the computation back to what it was before the action began executing. The revised model is as follows: now, if an action encounters a fault and attempts to abort it may either be rolled back or forced to completion. When forced to completion it has a choice of either terminating "normally" or "abnormally." In the revised model of computation, an exception handler has been posited for *Aeolus*.

In figure 7 action B is nested within action A. The recovery mechanism may terminate B normally or abnormally. Normal termination of B permits either

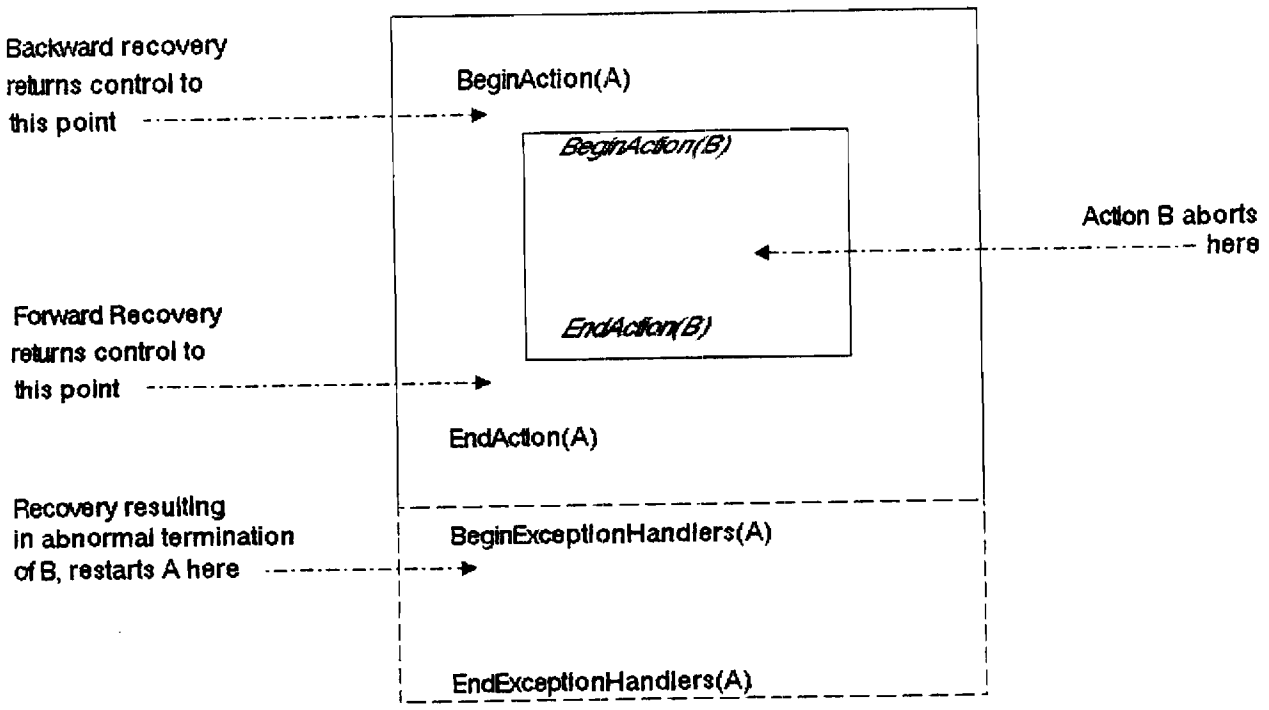


Figure 7. Resuming Computation Following Recovery

forward or backward recovery. Abnormal termination of of B causes control to pass to one of A's exception handlers.

B also may terminate either normally or abnormally even if it does not abort and then recover.

Under the revised model of computation the appropriate recovery strategy depends, in large measure, on where the fault is detected. An action may:

1. detect an error in its input. If an operation discovers error in its input data, it should be rolled back. The parent action, on being informed of the failure, should either attempt to fix the problem or otherwise compensate for the failure. If this is not possible it should fail itself;

2. detect an error in its own computation, i.e., an internal error. Even if an action reports an internal error, the parent action may decide the problem was incorrect input. In this case the parent action should proceed as above. If, however, the parent action agrees the problem was indeed an internal error, it should find an alternative means of accomplishing the task or at least attempt to contain the error and to provide a reduced level of service;

3. produce incorrect results. When the fault is discovered, there should be an attempt to prevent it from infecting the rest of the software and causing additional problems. This may involve setting some flags or otherwise making the fact of the failure visible in a consistent manner.

Backward error recovery returns control to the point just prior to the beginning of the action which failed, after constructing a state which satisfies the program invariant at that point. Forward error recover forces an action to complete in the sense that control passes to a point just following the end of the failed action. While forward error recovery may not provide an alternative means for carrying out the action, it should construct a state which permits the remander of the program to execute correctly. In some circumstances the recovery mechanism must not only construct a state which satisfies the invariant, it must also adjust that state to reflect changing conditions over which it has no control, e.g., the passing of time or the occurrence of an irreversible event. Irreversible events are usually understood to be physical events but may also refer to aspects of the state of the computation which cannot be modified by the recovery mechanism.

These strategies may be used to contain the consequences of the fault, repair the damage it caused before detection, and attempt to prevent the error from causing faults in the future. There are several approaches available:

1. "repairing" the data corrupted by the fault. The may involve restoring the old values, calculating new values, or setting flags which will tell other components not to trust the data and to do its calculations another way;

2. altering control information so that other components will not use or propagate the corrupted data, to prevent subsequent operations from executing the code containing the error, to force execution down paths which will compensate for the fault and its consequences;

3. in the extreme the recovery mechanism may calculate values and set control flags in preparation to cutting over to a backup system or a reduced level of service. In practice, this may be the most commonly used approach.

3.3 Degrees of Fault Tolerance

A large software system will almost certainly contain errors at the time it is placed into service. While software maintenance will remove those errors as they are uncovered, efforts to alter and enhance the software will inevitably introduce new errors. Software errors must be considered an inexhaustible source of faults and it is necessary to design large systems so as to tolerate a variety of software errors. This task is made difficult because the exact nature of the errors can not be anticipated by the programmer.[3] Often fault tolerant techniques cannot fully correct for the consequences of the error and must apply other strategies in an attempt to

3. If the programmer had a complete and accurate understanding of the errors in his program, he could fix them before the software is placed into service!

achieve less ambitious goals.

There are other sources of faults besides errors in the architecture and coding of the software. Unlike software, hardware components do wear out, and as they fail, they will inevitably produce a second class of faults.

A third class of faults will arise because of inadequacies in the models used to describe the systems which interact with the software—loads (e.g., transactions per second) may not be as expected, or a system may enter a state not anticipated by the software designers (e.g., a components in the physical system may interact in unexpected ways or may be forced to operated at or beyond designed capabilities).

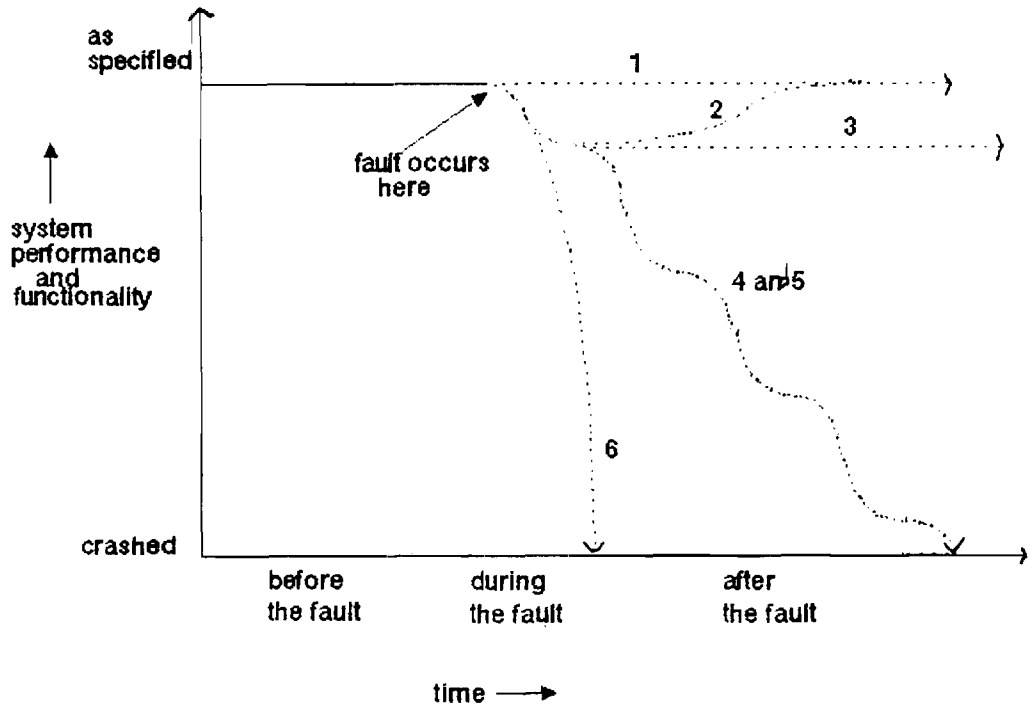


Figure 8. Degrees of Software Fault Tolerance

Figure 8 shows some of the possible fates which may await a software system following a fault. The numbers correspond to the degrees of fault tolerance listed below. Recovery may be complete (1 and 2), though 2 shows some temporary degradation in performance or functionality. Other times it may be necessary to allow some degree of long term degradation (3) in performance or functionality. While the fault may be detected, recovery may not be fully successful (4 and 5), and the resulting cascade of faults may result in progressive system failure. If the fault goes undetected (6), catastrophic failure may ensue.

Several degrees of software fault tolerance can thus be distinguished:

1. software fault is detected and corrected before system performance is impaired;
2. software fault is detected and compensated for. While anomalies in system performance may be detected, mission objectives are not jeopardized;

3. software fault is detected but cannot be corrected or compensated for. Recovery operations result in reductions in system performance but in a way which does not cause faults in other elements of the system. Human intervention may be required if all mission objectives are to be achieved;

4. software fault is detected and while it is contained and will not cause further software errors, elements of the associated physical systems may no longer be adequately controlled. Without human intervention, physical components may fail and this in turn may cause further software faults. Human intervention is required if primary mission objectives are to be achieved. Secondary mission objectives may be compromised;

5. software fault is detected but cannot be contained. A cascade of software faults will result in system failure. Primary mission objectives will not be met unless alternative command and control procedures are available;

6. software faults are not detected. Success of the mission depends on the correctness of the system and whether the structure of the software is such that the consequences of software faults will be adequately contained.

When human intervention is required, that intervention may take any of several forms. The human may assume some of responsibilities previously assigned to the software: this may require the human to interface with lower level software modules directly. In such a case "recovery" is the process of making a graceful transition from automatic to semi-automatic control and may involve shutting down the failed software system in a graceful manner and the proper initialization of displays and other elements of the human/computer interface.

Alternatively, the human may be responsible for "repairing" the faulting system. In this case the human must be presented with information about the fault and with the tools for carrying out the repair. A repair may involve reassigning the responsibilities of the faulty software component to other components capable of providing the necessary services, though perhaps in a less than optimal manner. The repair might also be a matter of adjusting some parameters or other state information and enabling the faulty component to resume operation. The ideal, but rather unlikely, repair would involve locating and fixing the software error during the mission.

In all cases, the recovery mechanism should log information about the fault so programmers can, after the mission, locate and fix the underlying error.

3.4 A Mechanism for Fault Tolerance

In environments such as *Clouds*, data is shared among applications by means of persistent objects. An object encapsulates both the data describing its state and the operations which reference or alter that state. These operations may also invoke operations on other, distinct objects.

Within *Clouds* and *Aeolus*, the associated programming language, a distinction is made between recoverable and nonrecoverable objects. This distinction provides a means for to incorporating irreversible operations into the *Clouds* model of computation. Interactions with physical systems can be embedded within nonrecoverable objects. The data area within such an object will include sensor registers (read only) and command registers (write only). Object entry points which write to command registers will be regarded as irreversible. As a pragmatic issue, these lowest level, irreversible operations should be very simple in structure. If the entry point which performs the irreversible operation must be complex, then it should contain simpler, nested actions which actually perform the irreversible operation.

Actually, not all writes to command registers should be regarded as irreversible. At the lowest level, irreversible operations cause a state change in a nonrecoverable object or, of interest here, in a physical system.

3.4.1 Actions The fundamental unit of work is the *action*. The notion of *action* described here is only partially implemented within *Clouds* and represents an extension of the notions underlying that system's design. If an action notices a fault has occcurred, it may attempt to abort. If an action faults, i.e., it divides by zero, and this is noticed by the runtime system, an attempt should be made to locate an appropriate exception handler within the action. If none is found, the runtime system should abort the action. The exception handler may also force the action to abort. When an action aborts, recovery is initiated. The recovery mechanism employs event handlers and exception handlers. The event handlers guarantee that certain minimal recovery steps are taken. The exception handlers, while providing no such guarantees, give the programmer an opportunity to employ some additional and more specialized steps as part of the recovery process. A detailed description of the recovery mechanism provided in the next subsection.

An action defines a sequence of operations on program data visible to it (in *Aeolus* static scoping rules are used) and on objects for which capabilities are known. An action should have simple, well defined semantics. By explicitly showing the boundaries of an action, the programmer makes it possible for the programming language's run time system, the operating system, and other sources of run time support to manage the action and its execution as a well defined unit. In particular, it provides a focus for such support activities as concurrency control and, of interest to the present discussion, fault tolerance. By adopting suitable programming conventions, it should be possible to mimic this extended notion of an action within the existing *Clouds* system.

Actions should be well defined both semantically and textually: the former makes it possible to support the action as a unit during execution, and the latter is of use to programmers who must construct or maintain programs which incorporate actions as a design element.

1. *Semantically well defined* means the action performs a single task which is easily described within the overall design of the program; when an action is semantically well defined, a programmer should be able to construct invariants characterizing the states of the computation both before and after the execution of the action. Invariants may "admit" to the possibility faults may occur, i.e., they may be of the form "if the action has failed, X must be true; if the action has succeeded, then Y must be true."

   If an action faults, the role of the recovery mechanism is to construct a consistent state, i.e., a state which satisfies a program invariant, and resume the computation at the point associated with that invariant. Backward recovery constructs a state which satisfies the invariant preceding the action and re-executes the action, perhaps using a different algorithm. Forward recovery constructs a state which satisfies the invariant following the action an resumes normal execution at that point in the code.

2. *Textually well defined* means the block of code defining an action is clearly marked with **BeginAction** and **EndAction** statements. These statements are matched statically and must be visible at the same level. Procedures or functions may also be declared to be actions, but this is not required. An object's entry points, (i.e., operations which may be invoked on an object) may be declared to be actions as well.

Actions, in our terminology, are *instantiated* each time they are executed. To facilitate action management in general and fault tolerance in particular each instance of an action is to have a unique name. A data area will be associated with each instance. This data area will be partitioned into attributes. Some of the attributes can be set by the programmer when defining the action and others can be set when an instance of the action is generated. The various routines involved in managing the execution of an instance of an action will use the attributes when deciding how to handle various situations arising at run time; additional attributes may be set to indicate certain situations were encountered during execution or certain decisions were made regarding the management of that instance of the action. The requirement that names be unique allows each instance of an action to be managed separately. There are, however, some technical details associated with this naming scheme.

The programmer will supply a local name for each block of code defining an action; these names must be unique within a given name space. The names will be extended by prefixing the names of the actions within which it is nested. If several instances of an action are generated from the same program text (e.g., an action is embedded in a loop) then the names of the instances will be distinguished by suffixing an instance number. These naming conventions will make it possible to support the concurrent execution of several actions even when they were generated by the same piece of code. Associating attributes with the names provides a means

by which an executing program can interact with the run-time system an other routines supporting its execution. In particular this mechanism can be used by the programmer to indicate how an action should be handled if it aborts. The routines supporting execution will know, for example, whether forward or backward recovery should be used. By setting the appropriate attributes it will be possible to indicate that recovery occurred, the methods used during recovery, and any other information the program may need to resume proper execution.

3.4.2 Event and Exception Handlers When an action aborts, it is important to preserve information about any irreversible operations it may have performed and to use that information in constructing an accurate and consistent state from which computation can be resumed.

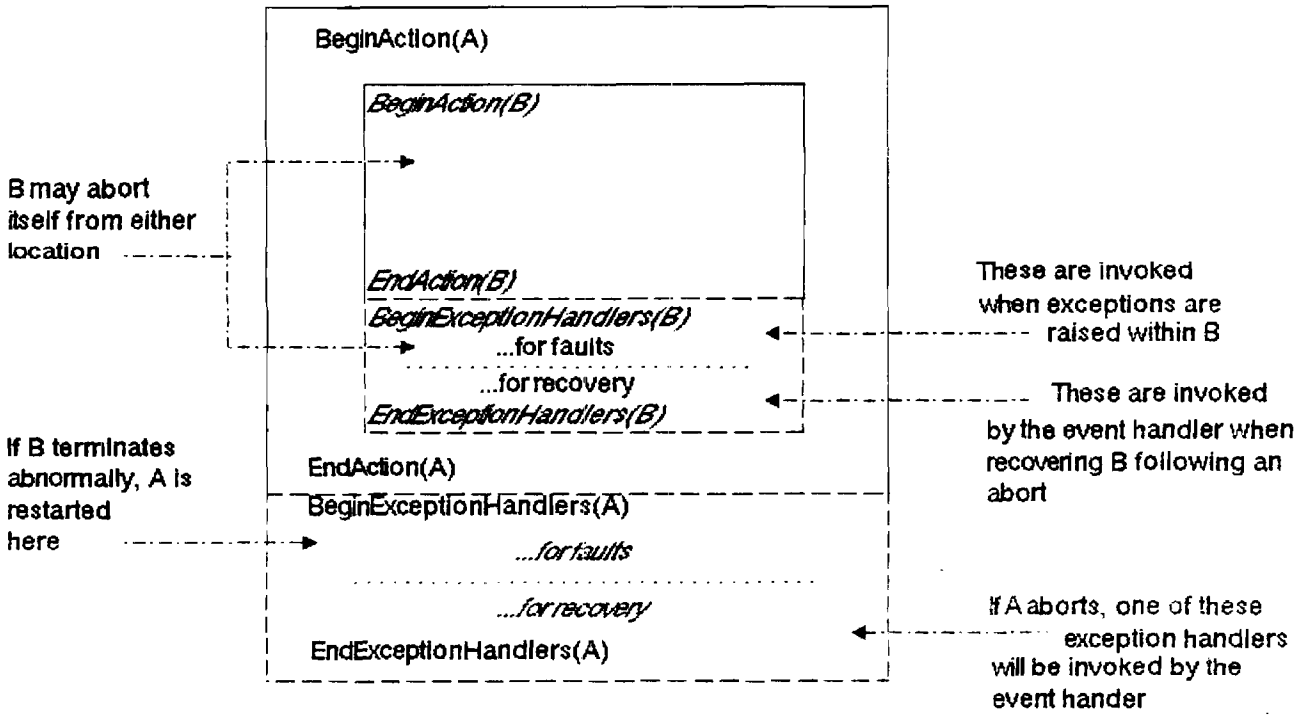


Figure 9. Exception Handlers and Event Handlers Work Together to Recover from a Fault

Some of this recovery can be done automatically but other aspects of it must be sensitive to the semantics of the action and the context in which it was executing. The process of initiating and carrying out recovery consists of four phases; this may be understood with reference to Figure 9.

1. Recovery is initiated when an action aborts itself or is aborted by the runtime system. An action may abort itself when it determines that a fault has occured. An action may be aborted by the runtime system if an exception is raised during the action's execution and an appropriate exception handler has not been defined within the action. Even if an exception handler is invoked, it may decide to abort the action.

2. An automatic recovery mechanism called an event handler should begin the recovery process. The event handler should be sensitive to the attributes of the action and may set some of the attributes to indicate the circumstances under which the abort (and associated fault) occurred. The event handler should be responsible for recording which irreversible operations were executed by the action and any other information essential for recovery.

3. The event handler has the option of passing control to an exception handler defined within the aborted action. This exception handler is provided by the programmer and should be designed to reconstruct the state of the computation. While this exception handler is responsible primarily for ensuring that the state of the computation reflects the occurrence of the irreversible operations in a consistent manner and terminating the action in a "clean" manner.

4. If the event handler is not able to pass control to an exception handler defined within the aborted action or if the exception handler is not able to terminate the action in a way which allows computation to continue immediately before or after the action itself, it should raise an exception visible to the parent action. This gives the parrent action an opportunity to further repair the state of the computation and greater control over where computation resumes. If an appropriate exception handler is not found withing the parent action, the parent action is forced to abort and another round of recovery is attempted.

Programmers often find it useful to separate alternative paths of execution into two groups: those which are available to normally executing programs and those which would be followed should various error conditions be detected. This distinction is made at a programmer's discretion and usually in an effort to clarify the structure of an algorithm. The three phases described above support a programmer in his effort to maintain this distinction in his work.

If an action faults and then aborts, the event handlers would, in the presence of irreversible operations, cancel the abort and attempt to terminate the action cleanly. An aborted action may still terminate normally if the event and exception handlers defined within it were able to find an appropriate continuation. Otherwise, an exception signaling abnormal termination would be propagated into the parent environment. The action is regarded as terminating abnormally, if an appropriate exception handler cannot be successfully invoked. Depending on circumstances then, the recovery activities may be carried out either by event and exception handers defined for the action or by the parent environment. In this latter case, the event and exception handlers must at least propagate enough information into the parent environment, for it to recognize and execute the proper continuation.

If an action contains several operations which are potentially irreversible, it is necessary to distinguish between those which have and have not occurred. An

attribute will be associated with each occurrence of a potentially irreversible operation. The attribute will serve as a flag indicating the execution state of the action. The flag is clear if its associated operation has been instantiated but has not become irreversible. The flag is set once the operation becomes irreversible. This is all done automatically by the event handler.

Some of the conditions discussed above present an unfortunate complication: it may become necessary for the recovery mechanism to deal with a partially completed, irreversible action, e.g., an irreversible action fails along with its associated exception handler and the exception handler in the calling environment. The system of naming actions and associating attributes with them, will allow higher level event and exception handlers to perform recovery by inspecting, perhaps recursively, the flags for the irreversible operations and actions within the hierarchy of nested actions.

The flags used to record the occurrence, or non-occurrence, of irreversible actions and operations may also be used to propagate that information into the calling environment. By this means, it is possible to provide the minimum amount of information required by an exception handler dealing with an abnormally terminated action.

Within *Aeolus* an event handler is invoked when an action aborts. Default event handlers are provided by *Aeolus*, but it is also possible for programmers to supply their own. A programmer supplied event handler is intended to be sensitive to the semantics of the action to which it is attached. If a programmer makes an error in coding an event handler for aborts, it may not be possible to contain the consequence of the fault. The alternative proposed here is to augment the event handler with an exception handler: the event handler has only limited functionality but provides a guarantee that it will execute without faulting, whereas the exception handler provides the programmer with greater control over the recovery but offers no guarantees about immunity from faults. The event handler will carry out the initial stages of recovery and will insure that information about irreversible operations is propagated outside the boundaries of the failed action. The event handler will be sensitive to the attributes of the action; indeed, the programmer's only means of specifying the semantics of the event handler will be by means of those attributes. While the event handler will ensure the minimum necessary information will survive an action's abort, the associated exception handler will have the initial responsibility for using that information to construct an appropriate state from which to resume computation.

3.4.3 Using the Mechanism We can distinguish a number of strategies for achieving fault tolerance simply by considering alternative mechanisms for constructing a state from which normal computation can be resumed.

I. Backward Recovery

A. Roll back using

1. recovery blocks

2. logging (e.g., undo/redo logs)

B. Roll back but exempting certain data areas

1. require that the programmer declare which data areas are exempt from roll back

2. let the decision as to which areas are exempt form roll back be made at the time recovery is attempted

C. Roll back, exempting certain data areas from roll back and doing some additional computation (e.g., event handlers, exception handlers) to complete the reconstruction of the state.

1. let the additional computation reference the rolled back state

2. let the additional computation reference only state as it was before roll back

3. let the additional computation reference both the states which preceded and following roll back.

II. Forward Recovery

A. Forward recovery by using event and exception handlers to complete the action's execution

B. Forward recovery by using event and exception handlers but allowing certain data areas to be rolled back to the state they had before the action began

1. controlling the decision as to what data areas will be rolled back

a. require that the programmer declare which data areas must be rolled back

b. let the decision as to which areas will be rolled back be made at the time recovery is attempted

2. decision as to what data may be referenced when computing recovered state

a. let the additional computation reference the recovered state

b. let the additional computation reference the state as it existed just prior to roll back.

c. let the additional computation reference both the states preceding and following roll back.

III. Transferring control to some location other than immediately before or after the action. This may be of some use in cutting over to a backup system or to a reduced level of service.

I.C and II.B are similar. In both the state is reconstructed through a combination of roll back and additional computation. The differences lie in where the computation is restarted, either before or after the failed action. The choice should be governed by considerations such as which invariant will be easier to satisfy and which makes more sense given the way in which the failure fits into the conceptual structure of the program. If the action was to fire a weapon and the failure resulted in a misfire, there are circumstances in which the appropriate action is to re-execute the action and others in which the misfire should be accepted as an outcome of the action. Equivalent software can be constructed using either approach. The choice should be made by those constructing the software, and they should be guided by a desire to keep the conceptual structure of the software as simple as possible.

For example, it is possible to let the main line of a program be aware that control reached a particular point following recovery; from a fault. In the case of the misfire example this is the proper course; the program invariants should be explicit about the handling of such events.

The misfire may have occurred for many reasons and and at many points in the process of controlling the weapon. Regardless of the point at which the failure occurred or the reason, the procedures for coping with it are similar. Fault tolerant constructs give the programmer a means of handling the failure in a way which does not burden the main fire control algorithm.

If we succeed in making fault tolerance inexpensive, a programmer can organize the software around his understandings of normal and abnormal execution.

3.5 Irreversible Operations and Fault Tolerance

Considerable thought must be directed towards the pragmatic issues related to using the mechanism described in the previous section. If recovery is to be done successfully, it is necessary to deal appropriately with any irreversible operations which may have occurred. This in turn requires the recovery mechanism recognize that an irreversible operation has occurred.

3.5.1 Recognizing Irreversible Operations Determining whether an operation is irreversible and working out the accompanying implications may not be a simple task. Operations may be actions (including object entries), procedures or functions.

1. At the lowest level, an irreversible operation is one which writes to a command register and thereby causes a physical system to change state or which changes

state information within a nonrecoverable object. These operations must be encapsulated within objects.

2. If the code for an action includes an irreversible operation, the action is said to be potentially irreversible.

3. If a potentially irreversible action is nested within a second action, the second action is also regarded as potentially irreversible.

4. A potentially irreversible action becomes irreversible as soon as it performs an irreversible operation or action.

All irreversible and potentially irreversible operations must be accompanied by some means for terminating the action in the event of failure. At the very least, it is necessary to preserve a record of which irreversible operations have occurred. This is provided by the mechanism described in the previous section.

In addition to the algorithmic problem of determing when a potentially irreversible operation becomes, in fact, irreversible, there is also the problem of defining the primitive, irreversible operations. Primitive, irreversible operations may occur in two ways:

1. operations may be performed on nonrecoverable objects; or

2. operations may trigger events in unrelated software systems or, more importantly, in the physical systems being controlled.

We can also distinguish several degrees of irreversibility:

1. operations which cannot be "undone" in the sense that they cannot be rolled back but for which an inverse operation exists;

2. operations which have no inverse but which are compensable;

3. operations which are not compensable.

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 has become irreversible. Even then it may be possible to stop, or reverse, the reaction by other means (e.g., removing the catalyst or adding other materials). As the reaction progresses, even such compensating actions may not be effective.

3.5.2 Sources of Faults in Irreversible Operations A general programming strategy will be to defer invoking an irreversible operation until all the possible software failure points have been passed. Also desirable to put guards on the irreversible

operation; these guards will check that the conditions for invoking the operation are indeed satisfied. We don't want to unnecessarily invoke an irreversible operation just because an ordinary, garden variety software bug, led us down that path or will keep us from finishing the job.

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

1. 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 operation itself;
2. the result of a sensor error. the sensor error may not be detected until after the irreversible operation is completed;
3. in the irreversible operation itself. The physical system being controlled may not respond to a command as expected. This may be because of influences on the physical system other than the control software; or
4. in an operation which uses the results of the irreversible operation but faults for other, unrelated reasons.

The choice of recovery strategy depends, in large measure, on the source and type of the fault. The next subsection presents some examples illustrating the range of alternative strategies available for different varieties of faults. The present subsection outlines many of the possible errors which may lead to faults as well as several ways in which faults may manifest themselves.

Faults may be:

1. transient or nontransient; or
2. the cumulative effect of a number of small errors or the result of a single error.

An error in the interface between control software and the system which in controls may be any of several types:

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

These 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.

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 resource, the use of the wrong resource, contention for a resource, a race condition in getting a resource, a resource not returned to 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 internal layer to provide these services in a consistent manner to external level. Errors are possible, especially in open systems or where the rules are complex, and fault tolerance in this area is required.

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 block or unblock on interrupts, the assumption that code is reentrant or not reentrant, the bypassing data interlocks, the failure to close or open an interlock, 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 the layout and management of data or about the propagation of control information.

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, inconsistent conventions as to the meaning of input or return values.

Faults may also be caused by errors in coding or in low level logic: a wrong operation may be used or missing all together; 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.

3.5.3 Recovering Irreversible Operations The purpose of fault tolerant computing is neither to fix nor even to precisely identify software errors. Rather, the objective is ensure the system continues to function, at a perhaps reduced level of service, in spite of any faults which may occur. While complete recovery may be possible in some or, perhaps, even in many cases, there will be many others in which there is no simple way around the code containing the software error. In such a case the recovery mechanism will be charged with cutting over, in a graceful manner, to a back-up system or to a reduced level of operation.

Since it is unreasonable for a programmer to anticipate all the possible faults in a software system, he should, instead, provide a recovery mechanisms which is appropriate to a broad class of errors or which will cope with any error, regardless of its type, within a particular block of code.

Once an action performs an irreversible operation, it must leave the computation in a state which reflects the occurrence of that operation. This can be assured through the proper use of the recovery mechanism. The recovery mechanism would also typically return control to a point just prior to or just following the action. It is also possible for the action to terminate "abnormally" and raise an exception in the calling environment. A taxonomy of programming techniques, software designs and operating philosophies associated with achieving fault tolerance in the presence of possibly irreversible operations is being developed. Several broad categories have been outlined:

1. to fix incorrect or corrupted data, or set some control flags which will direct execution down alternative paths thereby avoiding the software error which caused the fault when the code is next executed;
2. set up a "retry" using the same or an alternative block of code;
3. permit computation to continue as it would have had the fault not occurred (after adjusting data structures to reflect any irreversible consequence of the failed action);
4. perform the cutover to a backup system, to a reduced level of service or other, alternate mode of operation. Accomplishing this may require certain flags to be set and data structures modified.

This class of techniques is a generalization of the smaller class of techniques for achieving fault tolerance in the presence of only reversible operations. The first example illustrates some of these ideas in terms of an embedded software system which controls the firing of several weapons. These weapons have partially overlapping fields of fire. A second example involves printing a report, a task performed by a wide range of applications both within the military and industry. The second example illustrates one way in which a programmer may use the ideas related to recover in the face of irreversible actions to simplify the structure of his program.

3.5.4 Fire Control Example. One particular action, the fire control action, is responsible for aiming and firing the weapon. This action includes the code for operating the weapon and for recognizing any of several possible malfunctions. The action also has a means for tracking the particular target assigned to it. Under certain circumstances, the action is also responsible for producing a report describing the status of the weapon.

Another portion of the system is responsible for assigning targets to particular weapons. Initial target identification and tracking is performed by this part of the system. It continues to track a target until a particular weapon accepts responsibility for it. The target identification system posts information to an object shared by all the weapons. When a weapon needs a new target, it queries this object and

identifies what from its perspective is the highest priority target. For example targets which are within range of two weapons are of lower priority than a target within range of one weapon and whose trajectory will soon carry it out of range of any weapon.

Among the possible faults which might be encountered are a failure in a component or its associated driver, corruption of the object from which actions select their targets, and corruption of the data needed to determine target priorities.

Within the fire control action there are a number of operations which are irreversible or which may be conveniently regarded as such. The operation of firing at the target clearly becomes irreversible at some point. The operations involved in aiming the weapon may also be properly regarded as irreversible. After aborting the fire control action it is, in principle, possible to return the weapon to the position it had before that action began.[4] This, however, is not usually desired, especially if, once its been restarted, the action will resume tracking the same target. It is likely to be more efficient to simply ensure the recovery mechanism retains the new position of the weapon so it may be used by the fire control action once it is restarted.

This matter of retaining the position of the weapon across recovery is an important one. The fire control sequence may well involve a number of steps and invokes a number of operations on several different components of the weapon. Midway through a particular attempt at firing the weapon, one of the components may fail to respond. For simplicity, assume each operation on a component is encapsulated within an action which has the fire control action as its parent. When the component fails to respond to an operation, the nested action containing that operation aborts. The recovery mechanism for that nested action removes the component from service and cuts over to a backup system. For some components it may then simply be a matter of restarting the nested action which aborted and resuming the fire control sequence. For many important components with complex states and which are referenced a number of times during the fire control sequence, however, the simplest approach may be to abort the fire control action itself and restart the "count down." When the fire control action is restarted the current position of the weapon must be known.

A variation of this might arise if the component which failed did so in a way which corrupted some of the data describing the the position of the weapon. In this case, the recovery mechanism could restore the data by reading the appropriate

4. Clearly, the wrong approach would be to restore the data showing the old position while leaving the weapon in the new position.

sensors. Structuring recovery in this way means the main line of the fire control action need not know that it failed and need not concern itself with whether the position of the weapon has been correctly described.

A second variation might arise if the component fails only partially and no backup is available. The result might be a weapon which is operational but whose field of fire is now restricted. One strategy for handling this would be to abort the fire control action and, as part of recovery, update the tables used by the target selection algorithm when determining the relative priority of candidate targets. In the extreme, it may also be desirable to update the tables used by other weapons when selecting priority targets and, perhaps, to force other weapons to select new targets by aborting and restarting their fire control actions.

In the examples just considered, reference was made to an object shared by all of the weapons as well as the target identification system. A weapon needing a new target, consults this object and identifes what from the weapon's perspective is a high priority target. In order to meet timing constraints, it is possible that this object may have a somewhat complex structure. Suppose it contains a table candidate target table which shows most of the available targets, and perhaps a few invalid ones as well. A target is invalid if it has already been destroyed or has been selected by another weapon. It is also invalid if it cannot be found near the coordinates supplied by the target recognition process.

On selecting a target from the candidate target table, a weapon performs a few computationally simple checks to validate its selection. These checks might involve examining, for weapons with overlapping or neighboring fields of fire, selected and destroyed lists.[5] If the candidate target table does not contain any valid targets, a priority request is made of the target identification system to locate one, i.e., the use of the candidate target table speeds the system up but is not essential.

It is possible that from time to time the candidate target table becomes corrupted, or perhaps so full of invalid entries that it degrades rather than improves system performance. Suppose a fire control action aborts when it determines its selected target is invalid. As part of recovery, statistics are kept regarding the rate at which fire control actions abort because of invalid target selection. When this rate becomes excessive the runtime system has the option of aborting the target identification process (which is also an action, albeit a persistent one). As the target identification process recovers and restarts it should reinitalize the candidate target table, thereby lowering the rate at which fire control actions abort because of invalid

5. A background process is responsible for examing the destroyed lists and removing those targets from the candidate target table. It is also responsible for removing from the candidate target table any targets which have moved out of range.

target selection at least for a while. If statistics are kept by the recovery mechanism associated with the target identification process, it may be possible to recognize the existence of a fault within that process. When a fault is recognized, it may be possible to fix the problem by cutting over to a backup system for some component.

3.5.5 Information Processing Example. It may be desirable to regard some operations as irreversible even though they may, in principle be reversed if some tricky programming is employed. Suppose an action instatiated by an information processing system prints a status report just before it is aborted. The action is subsequently restarted. But what to do about the report? There are three ways of responding.

1. Rescind the order to print the report and let the action resubmit the report once the action itself is restarted. This may require some tricky programming and, if a printer has begun producing the report, may waste resources.

2. Restart the action and let it submit a duplicate report. This avoids the need for tricky programming but would still waste resources.

3. Restart the action but in a state that lets it know the print job has already been submitted. In other words, treat the submission of the print job as irreversible and make sure the fact it was submitted persists across recovery.

The third option seems the most attractive of the three, but it may be necessary to design a mechanism which will, in fact, guarantee the job is actually printed. With this guarantee, the action becomes even easier to code: not only may the programmer avoid the tricky recovery problems, but he neither need he concern himself with the task of monitoring the progress of the print job and resubmitting it if it is lost.

Ensuring that "print" is irreversible is a relatively simple task. When submitting the job, the material to be printed should be encapsulated within an object. The capability for that object should be sent to the appropriate printer. When the printer is ready for the job, an operation is invoked on the object and the material to be printed is returned. This is a fairly robust design and a good deal of fault tolerance may be incorporated into it.

Suppose the printer becomes inoperative. The printer driver can continue to invoke operations on the "print object." Each time an operation is invoked, it quickly aborts because the printer is down. This invokes the recovery mechanism within the print object. Each print object may recover in a different way. Some may choose to wait until this printer becomes available, others may reassign themselves to other printers, and some may adopt yet different stratagies.

If different printers require material be presented in different formats (e.g., as is the case with different brands of phototype setters) this can also be encorporated within the "print object" approach.

3.6 Summary and Directions for Future Work

A number of issues have been considered regarding irreversible operations and their implications for the design of applications in fault tolerant, object oriented systems. The fundamental problem is that of preserving information about irreversible operations which have been perform by an action when that action aborts and is recovered. A mechanism for performing this task has been described. A number of variations on this basic mechanism have been outlined. These variations need to be developed in greater detail. A syntax for programming language constructs supporting the basic mechanism and its variations must be developed. More careful consideration of various implementation issues is also required.

Problems related to recognizing the various types of irreversible operations have been considered. The reasons irreversible operations fault have also been discussed.

It has been recognized that it may not be possible to prevent a fault from effecting system performance or functionality. Consequently, various degrees of fault tolerance have been described and classified.

Deciding to achieve a desired degree of fault tolerance and then achieving it raises issues related to programming techniqes, software designs, and operating philosophy. These issues appear to be a rich area for further investigation. Additional work needs to be done in the area of developing paradigm examples illustrating how fault tolerance may be achieved and in developing a taxonomy of techniques.

Of particular interest is the possibility that "irreversible operations" should be regarded not as an obstacle to be "gotten around" but as a programming construct of considerable use in the design of real time systems—especially when used in conjunction with a powerful recovery mechanism. It may be possible to generalize on the present notion of an irreversible action: if a nested action is allowed to commit and make its results visible outside the parent action before the parent action itself commits, this nested action may well be called "irreversible" and treated using the techniques being developed in conjunction with the present project. This approach allows a "loophole" in otherwise atomic actions and is in sharp contrast with the philosopy adopted in *Clouds*. We believe, however, that the new approach deserves some careful consideration: it opens up the possibility for long lived actions which interact with their environment in complex ways and for employing weaker notions of serializability than can be enforced using locks alone. The weaker notions of serializability would be enforced instead by recovery mechanisms which are more sensitive to the semantics of the various actions and objects from which a

software system is constructed.

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). (To appear.)

[Ande81] Anderson, T., and P. A. Lee. *Fault Tolerance, Principles and Practice*. Englewood Cliffs, NJ: Prentice-Hall International, 1981.

[Birm85] Birman, K. P., and others. "An Overview of the ISIS Project." *Distributed Processing Technical Committee Newsletter* (IEEE Computer Society) 7, no. 2 (October 1985). (Special issue on Reliable Distributed Systems.)

[Birm85a] 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.)

[Dasg86] Dasgupta, P., and M. Morsi. "An Object-Based Distributed Database System Supported on the Clouds Operating System." Technical Report GIT-ICS-86/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986.

[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.)

[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.)

[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.)

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.

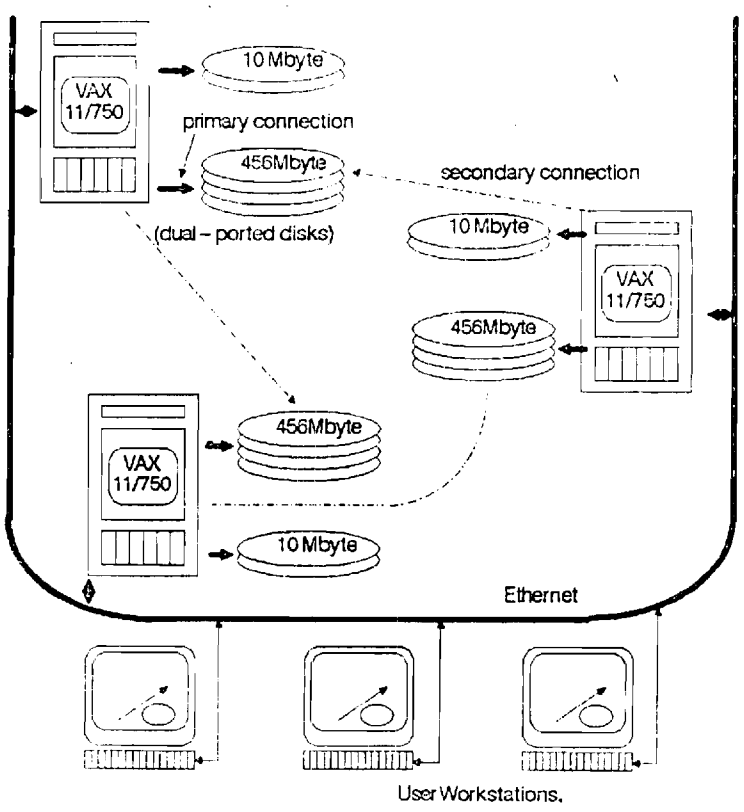


Figure 1. The Clouds Hardware Configuration

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.)

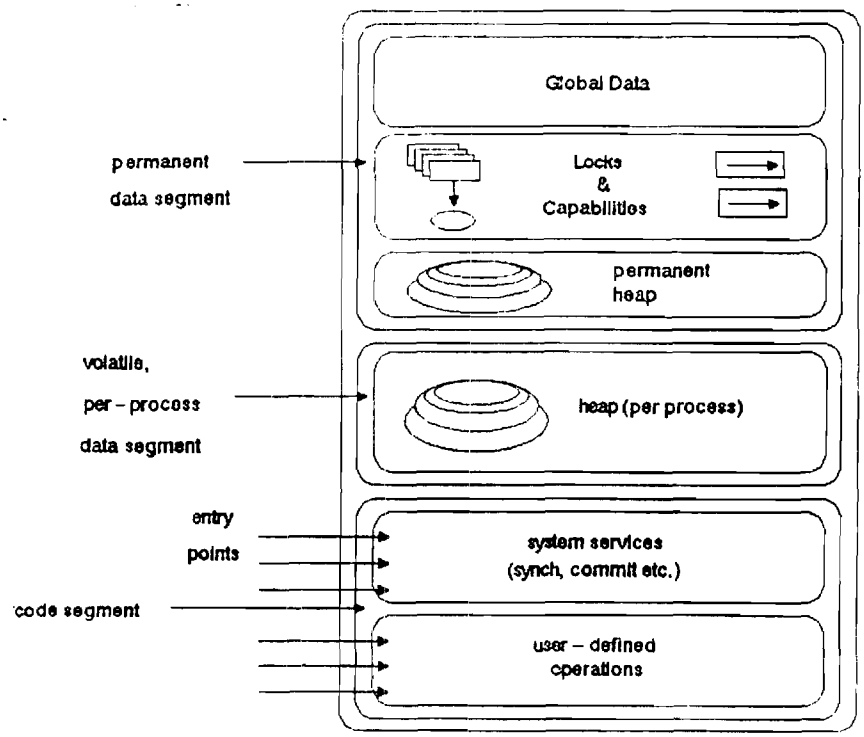


Figure 2. The Clouds Object Structure

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

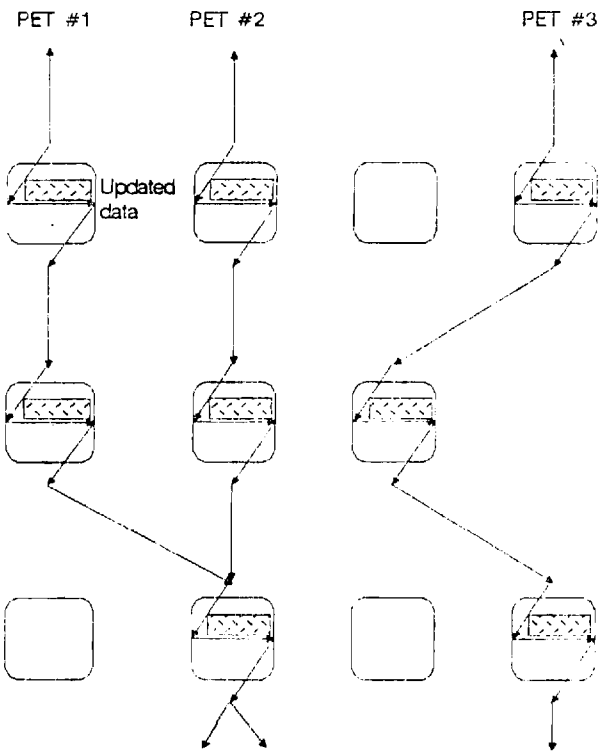


Figure 3. Parallel Execution Threads of 3-degree

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.

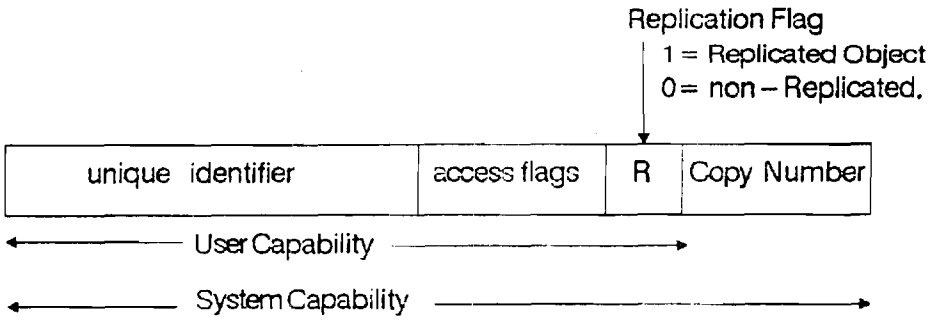


Figure 4. Capability Scheme for Replicated Objects

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.

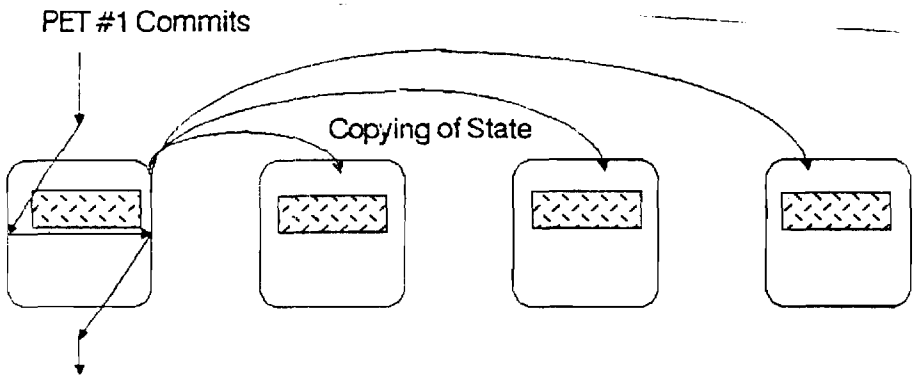


Figure 5. State Copying on PET Commit

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.)

```
definition of pseudo object Names is

   ! handles parsing of user names and command lines for OFS, shell

   MAXARGS : const unsigned := 32

   constraint argument_number is unsigned[ 0 .. MAXARGS ]

   MAX_PATH_LENGTH : const unsigned := 128

   constraint pathname_length is unsigned[ 0 .. MAX_PATH_LENGTH ]

   type pathname_type( length : pathname_length := 0 ) is new string( length )

   type path_type is ( in_ancestor, is_ancestor, in_child, in_here, is_here,
                       absolute_path, unknown_path )

operations

   procedure split
   (  path : in pathname_type(),
      first, rest : out pathname_type() )

    ! Separate the pathname given in parameter "path" into two
    ! parts, "first" and "rest", based on the first occurence of
    ! the path separator "/" in "path".


   procedure scan_path
   (  path : in pathname_type(),
      first, rest : out pathname_type() )
      returns path_type

    ! Performs the "split" operation (see above) on parameter
    ! "path", placing the results in the out parameters "first" and
    ! "rest", and returns the "path_type" of "path"


   procedure arg
   (  number : argument_number )
      returns pathname_type()

    ! Returns the argument the position of which in the argument
    ! string is given by "number".  The numbering of arguments
    ! begins with zero, where argument zero is (usually) the name
    ! of a process being invoked.


   procedure nargs ()
      returns argument_number

    ! Returns the number of arguments encountered in the current
    ! argument string.


   procedure get_args ()
```

```
    ! Reads an argument string from standard input and parses it
    ! into separate arguments for later use.

end definition.
```

```
definition of local object OFS
( OFS_ROOT : OFS,  initial_ancestor : OFS,  initial_name : string() ) is

   ! single-copy version of the Object Filing System

   import Names

   type OFS_op is ( mkdir_op, rmdir_op, mv_op, rm_op, ls_op, quick_ls_op )
      ! OFS operations which share essentially a common interface

operations

   ! In the operations below, the parameters first, rest, and ptype
   ! represent a pathname argument as processed by Names @ scan_path().

   procedure get_cap
   (  first, rest : in out pathname_type(),
      ptype : in out path_type,
      error : out boolean )
      returns capability examines

    ! Retrieves the capability for the object with the given
    ! pathname from the OFS.  If the pathname exists, sets error
    ! to FALSE and returns the associated capability, otherwise
    ! sets error to TRUE and returns NIL.


   procedure put_cap
   (  cap : capability,
      first, rest : in out pathname_type(),
      want_ancestor : boolean,
      ancestor : OFS,
      ptype : in out path_type,
      error : out boolean ) modifies

    ! Places a capability for an object into the OFS under the
    ! given pathname.  If the given pathname already exists, or if
    ! the prefix of the pathname (if any) does not exist, cap is
    ! not inserted into the OFS, and error is set to TRUE;
    ! otherwise, cap is inserted into the OFS, and error is set to
    ! FALSE.


   procedure general_op
   (  op_name : OFS_command,
      first_arg, second_arg : in out pathname_type(),
      error : out boolean ) modifies

    ! An interface for the operations enumerated by type OFS_op,
    ! which share a common interface and similar semantics with
    ! respect to pathnames.  The actual operation to be performed
    ! is specified by parameter op_name.  Two pathname parameters
    ! may be specified; the second is used only by the "mv"
    ! operation.  If the operation was successful (depending on the
    ! existence of the specified pathnames and, in the case of the
    ! rmdir command, on whether the first argument specified a
```

```
    ! directory), error is set to FALSE; otherwise, error is set to
    ! TRUE.


   procedure my_pathname ()
      returns pathname_type() examines

    ! Returns the current pathname of this OFS instance.


   procedure reset_pathname
   (  ancestor_name, new_name : pathname_type(),
      error : out boolean  ) modifies

    ! If parameter "ancestor_name" is the null string, sets the
    ! instance's current pathname to "new_name" and sets "error" to
    ! FALSE; else, if the concatenation of the ancestor's path and
    ! the new instance name is within the maximum path length
    ! contraint, sets the instance's current pathname to this
    ! concatenation, and sets "error" to FALSE; otherwise, sets
    ! "error" to TRUE.

end definition.
```

```
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 find ( name  : name_type   ,
                       error : out boolean ) returns value_type examines

         ! The FIND 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, FIND, 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.
```

```
implementation of object Names is

   ! handles parsing of user names for OFS

   import strings

   procedure split
   (!  path : in pathname_type,
    !  first, rest : out pathname_type !) is

      sep_pos : integer

   begin
      sep_pos := strings @ str_pos( path_separator, string( path ) )
      if sep_pos <= 0 ! no separator found ! then
         first := path
         rest := ''
      else
         first := pathname_type(
                     strings @ substr( string( path ), 1, sep_pos-1 )
                              )
         rest  := pathname_type(
                     strings @ substr( string( path ), sep_pos+1, path.length )
                              )
      end if
   end procedure ! split


   procedure scan_path
   (!  path : in pathname_type,
    !  first, rest : out pathname_type !)
    ! returns path_type ! is

      temp : pathname_type

   begin
      split( path, first, rest )
      if first = '/' ! a path relative to the root directory is desired ! then
         return absolute_path
      elsif first = '..' then
         if rest = '' ! the ancestor directory is desired ! then
            return is_ancestor
         else
            return in_ancestor
         end if
      elsif first = '.' then
         if rest = '' ! the current directory is desired ! then
            return is_here
         else
            return in_here
         end if
      else
         return in_child
      end if
   end procedure ! scan_path
```

```
   type arg_array( arg_length : pathname_length := 0 ) is
      array[ argument_number ] of pathname_type( arg_length )

   current_arg : arg_array()


   procedure arg
   (!  number : unsigned !)
    !  returns pathname_type() ! is

   begin
      return current_arg[ number ]
   end procedure ! arg


   numargs : argument_number := 0


   procedure nargs()
      returns unsigned is

   begin
      return numargs
   end procedure ! nargs


   arg_string : pathname_type()

   cur_index : pathname_length := 1


   procedure eat_separators
   ( done : out boolean ) is

   ! Consume all separators (blanks or tabs) beginning at the current
   ! position in the argument string.  The parameter 'done' is set
   ! to TRUE if the end of the argument string is encountered,
   ! FALSE otherwise.

   begin
      loop
         using next_char for arg_string[ cur_index ] do
            if ( next_char = ' ' ) or ( next_char = '\TAB\' )
              ! argument separator ! then
               cur_index += 1
            else
               done := next_char = '\NUL\'
               exit .
            end if
         end using
      end loop
   end procedure ! eat_separators


   procedure store_argument
```

```
   ( arg_number : argument_number ) is

   ! Parse an argument from the argument string and store it into
   ! the appropriate position in the array of parsed arguments.

      arg_index : pathname_length := 1

   begin
      loop
         using next_char for arg_string[ cur_index ] do
            if ( next_char = ' ' ) or ( next_char = '\TAB\' ) ! separator ! then
               exit .
            else
               current_arg[ arg_number ] [ arg_index ] := next_char
               arg_index, cur_index += 1
            end if
         end using
      end loop
      current_arg[ arg_number ] [ arg_index ] := '\NUL\'
   end procedure ! store_argument


   procedure get_args() is

      at_end_of_args : boolean

   begin
      InOut @ ReadStr( string( arg_string ) )
      for i in argument_number loop
         eat_separators( at_end_of_args )
         store_argument( i )
         if at_end_of_args  then
            numargs := i
            return .
         end if
      end loop
      numargs := MAXARGS
   end procedure ! get_args


   inithandler is

   begin
      null
   end inithandler

end implementation.
```

```
implementation of object OFS
!( OFS_ROOT : OFS,  initial_ancestor : OFS,  initial_name : string() )! is

   ! Single-copy version of Object Filing System.  OFS uses the
   ! symtab object type to manage all symbol table functions such
   ! as insertion, deletion, etc.  When an operation is invoked on
   ! an OFS object by a user, it is because that instance of OFS is
   ! the current working directory.  When a relative path is passed
   ! to that instance of the OFS, it relays the operation to the
   ! appropriate instance by invoking operations on its children
   ! (maintained in the symbol table) or its ancestor.  Absolute
   ! paths (i.e., paths starting at the root OFS) are not handled,
   ! except in put_cap(); it is assumed that the driver process
   ! (e.g., "shell") catches absolute paths and relays them to the
   ! root OFS.  (As an optimization, the driver should also catch
   ! most references to the current working directory's direct
   ! ancestor.)  The user may also change the instance of OFS which
   ! is considered the current working directory by means of
   ! relative or absolute paths.

   import symtab, InOut

   table : symtab( pathname_type(), capability ) := new symtab


   procedure get_cap
   (!  first, rest : in out pathname_type,
    !  ptype : in out path_type,
    !  want_ancestor : boolean,
    !  ancestor : out OFS,
    !  error : out boolean
   ) ! returns capability ! is

      child : OFS
      temp : pathname_type()

   begin
      error := FALSE
      ancestor := NIL
      case ptype of
         in_child, in_ancestor :
            child := table @ find( first, error )
            if error ! no child by this name exists ! then
               return NIL
            else
               temp := rest
               ptype := Names @ scan_path( temp, first, rest )
               return child @ get_cap( first, rest, ptype,
                                       want_ancestor, ancestor, error )
            end if
      || in_here, is_ancestor, is_here :
            if want_ancestor then
               ancestor := table @ find( '..', error )
            end if
            return table @ find( first, error )
      otherwise
```

```
      error := TRUE
      return NIL
   end case
end procedure ! get_cap


current_pathname : pathname_type()  ! the current name of this OFS instance


procedure put_cap
(!  cap : capability,
 !  first, rest : in out pathname_type(),
 !  ptype : in out pathname_type(),
 !  error : out boolean
) is

   child : capability
   child_OFS : OFS
   is_OFS : boolean
   temp : pathname_type()

begin
   error := FALSE
   case ptype of
      in_child, in_ancestor :
         child := table @ find( first, error )
         if error ! no child by this name exists ! then
            return .
         else
            error := not object_type( child, 'OFS' )
            if error then
               InOut @ WriteStr( 'OFS: "' )
               InOut @ WriteStr( string( first ) )
               InOut @ WriteStr( '" is not a directory.' )
               InOut @ WriteLn()
               return .
            else
               temp := rest
               ptype := Names @ scan_path( temp, first, rest )
               OFS( child ) @ put_cap( cap, first, rest, ptype, error )
            end if
         end if
   || in_here :
         is_OFS := object_type( cap, 'OFS' )
         if is_OFS then
            child_OFS := OFS( cap )
            temp :=  ! save old pathname in case of error
               child_OFS @ reset_pathname( current_pathname, first, error )
         end if
         if not error then
            table @ insert( first, cap, error )
            if error and is_OFS then   ! restore old pathname
               Void( child_OFS @ reset_pathname( '', temp, error ) )
            end if
         end if
   || absolute_path :
```

```
            OFS_ROOT @ put_cap( cap, first, rest, ptype, error )
         otherwise
            error := TRUE
            return .
         end case
      end procedure ! put_cap


      procedure general_op
      (!  op_name : OFS_op,
       !  first_arg, second_arg : pathname_type(),
       !  error : out boolean
      ) is

         child : capability
         child_OFS : OFS
         first, rest, temp : pathname_type()
         ptype : path_type

      begin
         error := FALSE
         ptype := Names @ scan_path( first_arg, first, rest )
         case ptype of
            in_child, in_ancestor :
               child := table @ find( first, error )
               if error ! child not found ! then
                  InOut @ WriteStr( 'OFS: "' )
                  InOut @ WriteStr( string( first ) )
                  InOut @ WriteStr( '", no such directory.' )
                  InOut @ WriteLn()
                  return .
               end if
               error := not object_type( child, 'OFS' )
               if error then
                  InOut @ WriteStr( 'OFS: "' )
                  InOut @ WriteStr( string( first ) )
                  InOut @ WriteStr( '" not a directory.' )
                  InOut @ WriteLn()
                  return .
               end if
               temp := rest
               ptype := Names @ scan_path( temp, first, rest )
               child @ general_op( op_name, first, rest, ptype, second, error )
         ¦¦ in_here :
               case op_name of
                  mkdir_op :
                     child := new OFS( MySelf(), first )
                     table @ insert( first, child, error )
               ¦¦ rmdir_op :
                     child := table @ find( first, error )
                     if error then
                        InOut @ WriteStr( 'rmdir: "' );
                        InOut @ WriteStr( string( first ) );
                        InOut @ WriteStr( '", no such directory.' )
                        InOut @ WriteLn()
                        return .
```

```
                end if
                error := object_type( child, 'OFS' )
                if error then
                   InOut @ WriteStr( 'rmdir: "' );
                   InOut @ WriteStr( string( first ) );
                   InOut @ WriteStr( '" is not a directory.' )
                   InOut @ WriteLn()
                   return .
                end if
                table @ delete( first, error )
          || mv_op :
             child := table @ find( first, error )
             if not error then
                ptype := Names @ scan_path( second_arg, first, rest )
                put_cap( child, first, rest, ptype, error )
                if not error then
                   table @ delete( first, error )
                end if
             end if
          || rm_op :
             child := table @ find( first, error )
             if not error then
                if object_type( child, 'OFS' ) then
                   InOut @ WriteStr( 'rm: "' );
                   InOut @ WriteStr( string( first ) );
                   InOut @ WriteStr( '" is a directory.' )
                   InOut @ WriteLn()
                   error := TRUE
                else
                   table @ delete( first, error )
                end if
             end if
          || ls_op :
             table @ quick_list()
          || quick_ls_op :
             table @ exact_list()
          end case
       otherwise
          error := TRUE
          return .
       end case
    end procedure ! general_op


    procedure my_pathname ()
    ! returns pathname_type() ! is
    begin
       return current_pathname
    end procedure ! my_pathname


    procedure reset_pathname
    (!  ancestor_name, new_name : pathname_type(),
     !  error : out boolean
    )! returns pathname_type() ! is
```

```
      old_pathname : pathname_type()

   begin
      if ancestor_name = '' then
         current_pathname := new_name
         error := FALSE
         return ''
      end if
      ancestor_name :=
         pathname_type( strings @ str_concat( string( ancestor_name ), '/' ) )
      error := ( strings @ str_len( string( ancestor_name ) ) +
                 strings @ str_len( string( new_name      ) ) )
               > MAX_PATH_LENGTH
      if error then
         InOut @ WriteStr( 'OFS: instance "' )
         InOut @ WriteStr( string( new_name ) )
         InOut @ WriteStr( '", pathname too long.' )
         InOut @ WriteLn()
         return ''
      else
         old_pathname := current_pathname
         current_pathname := pathname_type(
            strings @ str_concat( string( ancestor_name ), string( new_name ) )
                                  )
         return old_pathname
      end if
   end procedure ! reset_pathname


   inithandler is
      ! initialize the symbol table to hold the current and ancestor OFSs

      error : boolean

   begin
      table @ insert( '.', MySelf(), error )
      if error then
         InOut @ WriteStr( 'OFS: instance "' )
         InOut @ WriteStr( string( initial_name ) )
         InOut @ WriteStr( '" cannot insert "."' )
         InOut @ WriteLn()
      end if
      if initial_name = '/' then   ! this is the root OFS
         current_pathname := initial_name
         table @ insert( '..', MySelf(), error )
      else
         reset_pathname( initial_ancestor @ my_pathname(), initial_name, error )
         table @ insert( '..', initial_ancestor, error )
      end if
      if error then
         InOut @ WriteStr( 'OFS: instance "' )
         InOut @ WriteStr( string( initial_name ) )
         InOut @ WriteStr( '" cannot insert ".."' )
         InOut @ WriteLn()
      end if
   end inithandler
```

```
end implementation.
```

```
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 read 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 ( write : [write] ,
                       read  : [read]  )


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, write )
      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, write )
   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

   begin
      Await_Lock( symtable_lock, write )
      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, write )
   end procedure ! delete


procedure find (! name  : name_type    ,
                ! error : out boolean !) ! returns value_type ! is

   ! The FIND 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, read )
      error := not symtable[ bucket_num ] @ find( name, value )
      Release_Lock( symtable_lock, read )
      return value
   end procedure ! find


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,
   ! FIND, 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, read )
      quick_list()
      Release_Lock( symtable_lock, read )
   end procedure ! exact_list


inithandler is

   ! Here, we initialize the symbol table.

   begin
      for i in hash_range loop     ! each bucket is initially empty
         region symtable[ i ] do
            symtable[ i ] := new bucket_list
         end region
      end loop
   end inithandler

end implementation.
```

```
process shell is

   ! simple shell prototype to demonstrate use of OFS

   import OFS, ProcessManager, Names, InOut

   type shell_command is ( activate, bye, mkdir, rmdir, mv, rm, pwd, ls, qls )

   prompt : string( 80 ) := 'Clouds> '

   OFS_ROOT : const OFS := new OFS( NIL, NIL, '/' )

      ! The root object of the Object Filing System.  The above
      ! initialization is used for the test version of the shell
      ! process, where OFS is declared as a local object.  In the
      ! "production" version, where OFS is a nonrecoverable
      ! (Clouds) object, the initialization would take the form:
      !
      !    OFS( capability{ sysname{ 0, 16#70f, 'B', 1 },
      !                     access_rights( 16#ffffffff ) } )
      !
      ! that is, an explicit construction of a capability (which,
      ! in the case of the OFS root, must be well-known).
      ! (Actually, the capability shown above is for the current
      ! flat-name-space nameserver, but serves to show the format
      ! for specification of a capability.)


   procedure get_cmd
   (  cmd : out shell_command,
      first, rest : out pathname_type(),
      ptype : out path_type ) is

    ! Issues a prompt, then invokes the Names object to parse the
    ! next argument string from standard input.  The zero-th
    ! argument is examined to see if it matches one of the shell
    ! commands (this is done by linear search in this prototype).
    ! If the argument is not a shell command, it is assumed to name
    ! a process residing in the OFS.

      cmd_str : pathname_type()

   begin
      InOut @ WriteStr( prompt )
      Names @ get_args()
      cmd_str := Names @ arg( 0 )
      Names @ scan_path( cmd_str, first, rest, ptype )
      if ptype = in_here ! possibly a shell command ! then
         for trial_cmd in shell_command[ bye .. qls ] loop
            if cmd_str = pathname_type( trial_cmd ) then
               cmd := trial_cmd
               return .
            end if
         end loop
      end if
      cmd := activate  ! if cmd_str doesn't match any shell command
```

```
   end procedure ! get_cmd


   ancestor, current : OFS := OFS_ROOT

   still_processing : boolean := TRUE


   procedure get_activatee
   (  first, rest : in out pathname_type(),
      ptype : in out path_type )
      returns ProcessManager is

    ! If a ProcessManager with the given pathname (as processed by
    ! Names @ scan_path()) is found, a capability to it is
    ! returned; otherwise, NIL is returned.

      temp : pathname_type()
      error : boolean := FALSE
      dummy : OFS

   begin
      case ptype of
         in_ancestor :
            temp := rest
            ptype := Names @ scan_path( temp, first, rest )
            cap := ancestor @ get_cap( first, rest, ptype, FALSE, dummy, error )
      || in_child, in_here :
            cap := current @ get_cap( first, rest, ptype, FALSE, dummy, error )
      || absolute_path :
            cap := OFS_ROOT @ get_cap( first, rest, ptype, FALSE, dummy, error )
      otherwise
         error := TRUE
      end case
      if error then
         InOut @ WriteStr( 'shell: "' )
         InOut @ WriteStr( string( first ) )
         InOut @ WriteStr( '" process not found.' )
         InOut @ WriteLn()
         return NIL
      end if
      error := not object_type( cap, 'ProcessManager' )
      if error then
         InOut @ WriteStr( 'shell: "' )
         InOut @ WriteStr( string( first ) )
         InOut @ WriteStr( '" is not a process.' )
         InOut @ WriteLn()
         return NIL
      else
         return ProcessManager( cap )
      end if
   end procedure ! get_activatee


   procedure get_invokee
   (  ptype : path_type )
```

```
      returns OFS is

    ! This procedure provides a bit of an optimization at the shell
    ! process level by determining from the pathname type whether
    ! to initially invoke the current OFS, its ancestor, or the
    ! root OFS.  The capability of the correct invokee is returned
    ! (or NIL, if "ptype" has a bad value).

   begin
      case ptype of
         in_ancestor, is_ancestor :
            return ancestor
      ¦¦ in_child, in_here, is_here :
            return current
      ¦¦ absolute_path :
            return OFS_ROOT
      otherwise
         return NIL
      end case
   end procedure ! get_invokee


   procedure get_dir
   (  name : pathname_type() ) is

      first, rest : pathname_type()
      temp_current, temp_ancestor : OFS
      error : boolean

   begin
      Names @ scan_path( name, first, rest, ptype )
      temp_current :=
         invokee @ get_cap( first, rest, ptype, TRUE, temp_ancestor, error )
      if error then
         InOut @ WriteStr( 'chdir: "' )
         InOut @ WriteStr( name )
         InOut @ WriteStr( '" not found.' )
         InOut @ WriteLn()
      elsif not object_type( temp_current, OFS ) then
         InOut @ WriteStr( 'chdir: "' )
         InOut @ WriteStr( name )
         InOut @ WriteStr( '" is not a directory.' )
         InOut @ WriteLn()
      else
         current := temp_current
         ancestor := temp_ancestor
      end if
   end procedure ! get_dir


   procedure arg_number_ok
   (  cmd : shell_command )
      returns boolean is

    ! Check the number of arguments provided with a command to see
    ! if it is correct.  This is a very simple-minded prototype, in
```

```
    ! that no option strings are allowed.

      arg_num : argument_number

   begin
      arg_num := Names @ nargs()
      case cmd of
         activate :
            return TRUE    ! may have any number of arguments
      || bye, pwd :
            return arg_num = 1
      || mkdir, rmdir, rm, ls, qls :
            return arg_num = 2
      || mv :
            return arg_num = 3
      end case
   end procedure ! arg_number_ok


   procedure process_cmd
    ( cmd : shell_command,
      first, rest : in out pathname_type(),
      ptype : in out path_type ) is

    ! Activate a process residing in the OFS, or process a shell
    ! command by invoking an OFS operation, based on the value of
    ! the parameter "cmd".  The parameters "first", "rest", and
    ! "ptype" should initially contain the zero-th argument from
    ! the argument string, as processed by Names @ scan_path().

      activatee : ProcessManager
      invokee : OFS
      op : OFS_op
      second : pathname_type() := ''
      error : boolean

   begin
      if not arg_number_ok( cmd ) then
         InOut @ WriteStr( string( cmd ) )
         InOut @ WriteStr( ': incorrect number of arguments.' )
         InOut @ WriteLn()
         return .
      end if
      case cmd of
         activate :
            activatee := get_activatee( first, rest, ptype )
            if activatee <> NIL then
               activatee @ activate()
            end if
      || bye :
            still_processing := FALSE
      || mkdir, rmdir, mv, rm, ls, qls :
            invokee := get_invokee( ptype )
            if invokee <> NIL then
               case cmd of
                  mkdir : op := mkdir_op
```

```
                || rmdir : op := rmdir_op
                || mv    : op := mv_op   second := Names @ arg( 2 )
                || rm    : op := rm_op
                || ls    : op := ls_op
                || qls   : op := qls_op
                end case
                invokee @ general_op( op, Names @ arg( 1 ), second, error )
            end if
        || chdir :
            invokee := get_invokee( ptype )
            if invokee <> NIL then
               get_dir( Names @ arg( 1 ) )
            end if
        || pwd :
            InOut @ WriteStr( string( current @ my_pathname() ) )
            InOut @ WriteLn()
        otherwise
           InOut @ WriteStr( 'shell: Invalid command "' )
           InOut @ WriteStr( string( first ) )
           InOut @ WriteStr( '".' )
           InOut @ WriteLn()
        end case
    end procedure ! process_cmd

begin
    while still_processing loop
       get_cmd( cmd, first, rest, ptype )
       process_cmd( cmd, first, rest, ptype )
    end loop
end process.
```

```
definition of recoverable object symtab
( name_type : type,  value_type : type ) is

! Single-copy symbol table object using the Aeolus/Clouds lock
! mechanisms for synchronization.   This Clouds object type provides
! a resilient implementation of the symbol table.
!
! 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 find ( name  : name_type   ,
                       error : out boolean ) returns value_type examines

         ! The FIND 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, FIND, 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.
```

```
definition of recoverable object OFS
( OFS_ROOT : OFS,  initial_ancestor : OFS,  initial_name : string() ) is

   ! Single-copy version of the Object Filing System.  This Clouds
   ! object provides a resilient implementation of the OFS.

   import Names

   type OFS_op is ( mkdir_op, rmdir_op, mv_op, rm_op, ls_op, quick_ls_op )
      ! OFS operations which share essentially a common interface

operations

   ! In the operations below, the parameters first, rest, and ptype
   ! represent a pathname argument as processed by Names @ scan_path().

   procedure get_cap
   (  first, rest : in out pathname_type(),
      ptype : in out path_type,
      error : out boolean )
      returns capability examines

    ! Retrieves the capability for the object with the given
    ! pathname from the OFS.  If the pathname exists, sets error
    ! to FALSE and returns the associated capability, otherwise
    ! sets error to TRUE and returns NIL.


   procedure put_cap
   (  cap : capability,
      first, rest : in out pathname_type(),
      want_ancestor : boolean,
      ancestor : OFS,
      ptype : in out path_type,
      error : out boolean ) modifies

    ! Places a capability for an object into the OFS under the
    ! given pathname.  If the given pathname already exists, or if
    ! the prefix of the pathname (if any) does not exist, cap is
    ! not inserted into the OFS, and error is set to TRUE;
    ! otherwise, cap is inserted into the OFS, and error is set to
    ! FALSE.


   procedure general_op
   (  op_name : OFS_command,
      first_arg, second_arg : in out pathname_type(),
      error : out boolean ) modifies

    ! An interface for the operations enumerated by type OFS_op,
    ! which share a common interface and similar semantics with
    ! respect to pathnames.  The actual operation to be performed
    ! is specified by parameter op_name.  Two pathname parameters
    ! may be specified; the second is used only by the "mv"
    ! operation.  If the operation was successful (depending on the
    ! existence of the specified pathnames and, in the case of the
```

```
    ! rmdir command, on whether the first argument specified a
    ! directory), error is set to FALSE; otherwise, error is set to
    ! TRUE.


   procedure my_pathname ()
      returns pathname_type() examines

    ! Returns the current pathname of this OFS instance.


   procedure reset_pathname
   (  ancestor_name, new_name : pathname_type(),
      error : out boolean  ) modifies

    ! If parameter "ancestor_name" is the null string, sets the
    ! instance's current pathname to "new_name" and sets "error" to
    ! FALSE; else, if the concatenation of the ancestor's path and
    ! the new instance name is within the maximum path length
    ! contraint, sets the instance's current pathname to this
    ! concatenation, and sets "error" to FALSE; otherwise, sets
    ! "error" to TRUE.

end definition.
```

Mar 1 17:23 1987 r_symtab.imp Page 1

```
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_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.

per action
   inserted : keyed_list( name_type, value_type ) := new keyed_list
   deleted  : list( name_type ) := new list
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


symtable_lock : lock ( write : [write] ,
                       read  : [read]  )

   ! The SYMTABLE lock allows the entire symbol table to be locked.
   ! This lock is set (in read 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.


name_lock : lock ( write : []     ,
                   read  : [read] ) domain is name_type

   ! 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.


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, write )
         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 write 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, write )
         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 find (! name  : name_type    ,
                ! error : out boolean !) ! returns value_type ! is
```

```
   ! The FIND 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, read )
      error := not sym_find( name, value )
      return value
   end procedure ! find


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,
   ! FIND, 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, read )
      quick_list()
   end procedure ! exact_list


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 @ append( Self.inserted )
         Parent.deleted  @ append( 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


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.
```

```
implementation of object OFS
!( OFS_ROOT : OFS,  initial_ancestor : OFS,  initial_name : string() )! is

   ! Single-copy version of Object Filing System.  This
   ! implementation of the OFS uses the recoverbility features of
   ! Clouds to provide resiliency.  OFS uses the symtab object type
   ! (which is also recoverable) to manage all symbol table
   ! functions such as insertion, deletion, etc.  When an operation
   ! is invoked on an OFS object by a user, it is because that
   ! instance of OFS is the current working directory.  When a
   ! relative path is passed to that instance of the OFS, it relays
   ! the operation to the appropriate instance by invoking
   ! operations on its children (maintained in the symbol table) or
   ! its ancestor.  Absolute paths (i.e., paths starting at the
   ! root OFS) are not handled, except in put_cap(); it is assumed
   ! that the driver process (e.g., "shell") catches absolute paths
   ! and relays them to the root OFS.  (As an optimization, the
   ! driver should also catch most references to the current
   ! working directory's direct ancestor.) The user may also change
   ! the instance of OFS which is considered the current working
   ! directory by means of relative or absolute paths.

   import symtab, InOut

   recoverable
      current_pathname : pathname_type()    ! current name of this OFS instance
   end recoverable

   table : permanent symtab( pathname_type(), capability ) := new symtab


   procedure get_cap
   (!  first, rest : in out pathname_type,
    !  ptype : in out path_type,
    !  want_ancestor : boolean,
    !  ancestor : out OFS,
    !  error : out boolean
   ) ! returns capability ! is

      child : OFS
      temp : pathname_type()

   begin
      error := FALSE
      ancestor := NIL
      case ptype of
         in_child, in_ancestor :
            child := table @ find( first, error )
            if error ! no child by this name exists ! then
               return NIL
            else
               temp := rest
               ptype := Names @ scan_path( temp, first, rest )
               return child @ get_cap( first, rest, ptype,
                                       want_ancestor, ancestor, error )
            end if
```

```
      !! in_here, is_ancestor, is_here :
            if want_ancestor then
               ancestor := table @ find( '..', error )
            end if
            return table @ find( first, error )
      otherwise
         error := TRUE
         return NIL
      end case
   end procedure ! get_cap


   procedure put_cap
   (!  cap : capability,
    !  first, rest : in out pathname_type(),
    !  ptype : in out pathname_type(),
    !  error : out boolean
   ) is

      child : capability
      child_OFS : OFS
      is_OFS : boolean
      temp : pathname_type()

   begin
      error := FALSE
      case ptype of
         in_child, in_ancestor :
            child := table @ find( first, error )
            if error ! no child by this name exists ! then
               return .
            else
               error := not object_type( child, 'OFS' )
               if error then
                  InOut @ WriteStr( 'OFS: "' )
                  InOut @ WriteStr( string( first ) )
                  InOut @ WriteStr( '" is not a directory.' )
                  InOut @ WriteLn()
                  return .
               else
                  temp := rest
                  ptype := Names @ scan_path( temp, first, rest )
                  OFS( child ) @ put_cap( cap, first, rest, ptype, error )
               end if
            end if
      !! in_here :
            is_OFS := object_type( cap, 'OFS' )
            if is_OFS then
               child_OFS := OFS( cap )
               temp :=  ! save old pathname in case of error
                  child_OFS @ reset_pathname( current_pathname, first, error )
            end if
            if not error then
               table @ insert( first, cap, error )
               if error and is_OFS then   ! restore old pathname
                  Void( child_OFS @ reset_pathname( '', temp, error ) )
```

```
            end if
         end if
      || absolute_path :
         OFS_ROOT @ put_cap( cap, first, rest, ptype, error )
      otherwise
         error := TRUE
         return .
      end case
   end procedure ! put_cap


   procedure general_op
   (!  op_name : OFS_op,
    !  first_arg, second_arg : pathname_type(),
    !  error : out boolean
   ) is

      child : capability
      child_OFS : OFS
      first, rest, temp : pathname_type()
      ptype : path_type

   begin
      error := FALSE
      ptype := Names @ scan_path( first_arg, first, rest )
      case ptype of
         in_child, in_ancestor :
            child := table @ find( first, error )
            if error ! child not found ! then
               InOut @ WriteStr( 'OFS: "' )
               InOut @ WriteStr( string( first ) )
               InOut @ WriteStr( '", no such directory.' )
               InOut @ WriteLn()
               return .
            end if
            error := not object_type( child, 'OFS' )
            if error then
               InOut @ WriteStr( 'OFS: "' )
               InOut @ WriteStr( string( first ) )
               InOut @ WriteStr( '" not a directory.' )
               InOut @ WriteLn()
               return .
            end if
            temp := rest
            ptype := Names @ scan_path( temp, first, rest )
            child @ general_op( op_name, first, rest, ptype, second, error )
      || in_here :
            case op_name of
               mkdir_op :
                  child := new OFS( MySelf(), first )
                  table @ insert( first, child, error )
            || rmdir_op :
                  child := table @ find( first, error )
                  if error then
                     InOut @ WriteStr( 'rmdir: "' );
                     InOut @ WriteStr( string( first ) );
```

```
                  InOut @ WriteStr( '", no such directory.' )
                  InOut @ WriteLn()
                  return .
               end if
               error := object_type( child, 'OFS' )
               if error then
                  InOut @ WriteStr( 'rmdir: "' );
                  InOut @ WriteStr( string( first ) );
                  InOut @ WriteStr( '" is not a directory.' )
                  InOut @ WriteLn()
                  return .
               end if
               table @ delete( first, error )
         || mv_op :
            child := table @ find( first, error )
            if not error then
               ptype := Names @ scan_path( second_arg, first, rest )
               put_cap( child, first, rest, ptype, error )
               if not error then
                  table @ delete( first, error )
               end if
            end if
         || rm_op :
            child := table @ find( first, error )
            if not error then
               if object_type( child, 'OFS' ) then
                  InOut @ WriteStr( 'rm: "' );
                  InOut @ WriteStr( string( first ) );
                  InOut @ WriteStr( '" is a directory.' )
                  InOut @ WriteLn()
                  error := TRUE
               else
                  table @ delete( first, error )
               end if
            end if
         || ls_op :
            table @ quick_list()
         || quick_ls_op :
            table @ exact_list()
         end case
   otherwise
      error := TRUE
      return .
   end case
end procedure ! general_op


procedure my_pathname ()
! returns pathname_type() ! is
begin
   return current_pathname
end procedure ! my_pathname


procedure reset_pathname
(! ancestor_name, new_name : pathname_type(),
```

```
    !  error : out boolean
   )! returns pathname_type() ! is

      old_pathname : pathname_type()

   begin
      if ancestor_name = '' then
         current_pathname := new_name
         error := FALSE
         return ''
      end if
      ancestor_name :=
         pathname_type( strings @ str_concat( string( ancestor_name ), '/' ) )
      error := ( strings @ str_len( string( ancestor_name ) ) +
                 strings @ str_len( string( new_name      ) ) )
               > MAX_PATH_LENGTH
      if error then
         InOut @ WriteStr( 'OFS: instance "' )
         InOut @ WriteStr( string( new_name ) )
         InOut @ WriteStr( '", pathname too long.' )
         InOut @ WriteLn()
         return ''
      else
         old_pathname := current_pathname
         current_pathname := pathname_type(
            strings @ str_concat( string( ancestor_name ), string( new_name ) )
                                  )
         return old_pathname
      end if
   end procedure ! reset_pathname


   inithandler is
      ! initialize the symbol table to hold the current and ancestor OFSs

      error : boolean

   begin
      table @ insert( '.', MySelf(), error )
      if error then
         InOut @ WriteStr( 'OFS: instance "' )
         InOut @ WriteStr( string( initial_name ) )
         InOut @ WriteStr( '" cannot insert "."' )
         InOut @ WriteLn()
      end if
      if initial_name = '/' then   ! this is the root OFS
         current_pathname := initial_name
         table @ insert( '..', MySelf(), error )
      else
         reset_pathname( initial_ancestor @ my_pathname(), initial_name, error )
         table @ insert( '..', initial_ancestor, error )
      end if
      if error then
         InOut @ WriteStr( 'OFS: instance "' )
         InOut @ WriteStr( string( initial_name ) )
         InOut @ WriteStr( '" cannot insert ".."' )
```

```
        InOut @ WriteLn()
      end if
   end inithandler

end implementation.
```