

FINAL REPORT  
RESEARCH ON RELIABLE DISTRIBUTED  
COMPUTING  
CONTRACT #MDA 904-84-C-6035  
REPORTING PERIOD: 11 Sept 84 - 30 Sept 85

## TABLE OF CONTENTS

	Page
1. Summary of Work Done	1
2. Distributed File Systems	2
2.1. Storage Management Design for an Action-based Operating System	2
2.2. Storage Management	5
2.3. Recovery Management and Virtual Memory	5
2.4. Storage Management Implementation	7
2.4.1. References	10
3. Language Support for Robust Distributed Programs	11
3.1. The Design of Aeolus	11
3.1.1. Features for Systems Programming	11
3.1.2. Features for Object and Action Programming	13
3.2. Programming Methodology	15
3.3. Implementation	16
3.4. Design Refinements	18
3.5. References	18
4. Conclusions	19
Funds Expenditure Graph	21
Funds Expenditure Report	22

## 1. Summary of Work Done

During the course of this project, a variety of work has done on the two tasks called for by the statement of work. These efforts are closely related to other work in progress within the Clouds Project, a major research effort in the School of Information and Computer Science of Georgia Tech in the area of reliable distributed computing. We are now very close to integrating the results of a number of individual tasks supported under this and other projects to produce a first prototype of the Clouds distributed computing system.

Under the Distributed File Systems task, work has concentrated on the development of the storage management system of the Clouds kernel. Since the fundamental concept of the Clouds system is to support transparent, reliable access to arbitrary objects located anywhere with a network multicomputer, distributed file system issues must be addresses at a much lower level than is traditionally the case. The major achievements of this task are as follows:

- (1) initial design of the storage management system (see quarterly progress report 1 and Appendix A to that report)
- (2) refinements to that design for improved efficiency (see quarterly progress report 2)
- (3) design and implementation of low-level device drivers to support Clouds reliability mechanisms (see quarterly progress reports 3 and 4)
- (4) integration of storage with kernel virtual memory management (see quarterly progress report 4 and Appendix A to that report).

The technical report "Notes on a Storage Manager for the Clouds Kernel" (appendix A to quarterly progress report 4), thoroughly documents the results our work on the Clouds Storage Manager.

Under the Language Support for Robust Distributed Programs task, the focus of our work has been our language, Aeolus, and its intended uses. The most important aspect of Aeolus is that it provides a high-level language interface to the action and object management features of the Clouds Kernel. Thus it is intended to be used as a systems programming language for the implementation of the layers of the Clouds

system above the kernel. In support of this implementation work, we have been studying programming methodology issues involving the use of these unique capabilities provided by the kernel. The major achievements of our language work have been:

- (1) completion of the initial design of Aeolus (see quarterly progress report 1 and Appendix A to quarterly progress report 3)
- (2) substantial progress on implementation of an Aeolus compiler (see quarterly progress reports 2 and 3)
- (3) initial programming methodology studies (see quarterly progress report 1 and 2 and Appendices B and A to those reports, respectively)
- (4) definition of the interface to kernel action and object managers (see quarterly progress report 4)
- (5) refinements of the Aeolus design (see quarterly progress report 4 and Appendix B to that report).

## **2. Distributed File Systems**

### **2.1. Storage Management Design for an Action-based Operating System**

The Clouds Project is an effort to provide support for a distributed computing system which achieves performance improvements (over conventional computing systems) through the parallelism possible in a multi-computer environment and reliability improvements through the redundancy available in processing resources and data storage. In order to achieve such improvements, the system must ensure the proper coordination of processes on various machines in the system and synchronize the use of shared data. The system as a whole must be able to deal with failures of one of its component machines, determining those processes on the failed machine which are necessary for the continuation of some larger task. A reliable distributed system must be able to ensure the consistency of data in the presence of machine failures, taking into account that data may be replicated.

The initial goal of the Clouds project is to produce an operating system kernel that provides the mechanisms needed by a reliable distributed computing system. In providing these mechanisms, the Clouds kernel must support other conventional

mechanisms such as virtual memory, process control and secondary storage management. The action and object support must be integrated with the conventional kernel functions so that support for a reliable distributed computing system is available through a well-defined kernel interface, and the implementation of the kernel is efficient and compact.

One subtask currently in progress is the design and investigation of a portion of the Clouds kernel: the *storage management system*. A Ph.D. research proposal by Pitts (attached as Appendix A to quarterly Progress Report 1) described the initial plans for work on this subtask. In addition to describing how such a system can be built, it also defined the interaction of the storage management system with other parts of the kernel, particularly its interaction with the virtual memory system.

Thus the purpose of this research task has been to design a kernel-level storage management (file) system (*storage manager*) that supports a reliable distributed computing system. The storage manager is responsible for the secondary storage available on the system. Specifically, the design presents the structures and mechanisms necessary to support the storage manager. The design includes support for both recoverable and non-recoverable objects. Mechanisms to create, delete, write and read objects on disks are defined. For recoverable objects the additional protocols and structures ensure recoverability of objects in the presence of machine failures. The design also discusses the interaction of the storage manager and the virtual memory system. This portion of the design specifies the structures and mechanisms required for virtual memory. The design also defines the support required for action management and object recovery. Finally a facility for the location of segments on secondary storage must be provided.

The design of the storage manager is being done two phases. Phase one has been a design of essential features for the system. The end-result will be an implementation for the Clouds kernel that will serve as a test-bed for further research. An analysis of the design and implementation will be done to determine the correctness and effectiveness of the design. The results of the analysis may have an effect on phase two. This phase of the design will include modifications and refinements to the original design. In general, phase two will include features not absolutely necessary for the storage manager, but which may be desirable later as the system is put to use as a research device. Feedback from the analysis of the original design may suggest

some of the changes found in the second phase. Phase two is not intended for immediate implementation.

The Clouds kernel will provide support for three basic mechanisms which will be important to later discussion: *processes*, *objects* and *actions*. Processes are the active agents of the system; to initiate and perform any work requires a process. The kernel has a process manager which handles all bookkeeping associated with creating, dispatching, and destroying processes.

Objects, on the other hand, are passive entities. Objects are typed collections of data. The type of an object determines what operations may be performed on the data, as well as how the data is organized. Object data can only be manipulated through these operations, and then only by a process which has a proper *capability* for the object. A capability is a unique name for an object along with a list of operations which are permissible for use by the possessor. The object manager handles the overhead of verifying capabilities and performing operation calls.

Objects are the organizational units of the system. By using objects, a programmer has a means for abstraction and isolation of data. The kernel also provides a mechanism for organizing sets of operations into a unit. This mechanism is the action. Actions are *atomic*. The set of operations comprising an action appears to execute completely (by *committing* its results) or not at all (by *aborting*). Also, the atomicity of actions prevents the execution of one action interfering with the execution of another. Actions provide a mechanism for making the effects of a set of operations consistent and recoverable.

Actions are managed by the object manager. Actions themselves are simply organizational units of work and require processes in order to perform any task. An action may have several processes or one process executing on its behalf.

The kernel provides processes, objects, and actions as efficiently as possible. Particularly, because objects have different types and different possible operations, the kernel needs access to objects in a manner which is consistent and convenient. For this reason, all objects have a secondary type, called the *segment* type. The segment type is a sequence of bytes with primitive operations such as read a page, write a page, and delete or add a page. The segment is accessible only by the kernel.

## 2.2. Storage Management

The Clouds secondary storage is managed as a set of *partitions*. Each partition is an autonomous logical device with its own set of interface routines for the transfer of information and the allocation of the secondary storage managed by the partition. A partition resides completely on one physical device and consists of a contiguous set of records on disk. The partition requires three structures to manage partition storage. First is a partition header, which holds information concerning the partition such as its size, whether it provides support for recoverability, a list of bad disk records for the partition, and other such information. The header should be duplicated to reduce the risk of its destruction by a media failure or other such disastrous error. The header is placed at a known location in the partition.

Each partition also maintains a directory, contains a mapping of sysnames (for objects) onto partition record addresses. Note that a partition directory contains mappings only for objects residing on that partition. Redundancy should also be insured for this structure. The partition directory is at a well-known location.

The third partition structure is a record map, which is a bit-map showing allocation of records for the partition. The driver uses the record map to determine which records are in use by segments and which can be allocated. Once again, the record map is an important structure which should be duplicated to prevent its loss after a media crash.

The remainder of the partition is available for the storage of object data, or as the storage manager treats objects, segment data.

Before a partition can be accessed by the kernel, it must be mounted on the system. This involves doing a consistency check on the partition storage, examining the directory and record map, and cleaning up any loose ends as far as recovery management is concerned. Of course, the physical device on which the partition resides must be active prior to this processing.

## 2.3. Recovery Management and Virtual Memory

Segment recovery is accomplished via a shadowing scheme. That is, segments on which actions are operating have shadow versions that the actions actually see. The

scheme is pessimistic, so that no modifications are made to a permanent version until the action making the modifications commits. The goals of the recovery scheme are, aside from producing consistent results, to allow recovery of segments (and partition structures) with as little storage overhead as possible, and with as few disks accesses as possible. Shadowing, then, will be minimal. That is, only those parts of the segment actually modified are shadowed.

The storage manager becomes involved in recovery only when a top-level action precommits and the shadow version of the segment on which the action is operating is created. Prior to precommit, all write operations are done in memory. An active segment is mapped into memory by the virtual memory system. An object's address space contains a block of permanent data and a block of volatile data. The permanent data block contains data which will survive a crash. This is basically the permanent object state. The volatile data block's contents will not survive a machine crash and generally consists of such structures as locks and semaphores for the object. Also contained in the volatile data block is much of the information maintained by the action management system.

When an action operates on a segment, the action management system maintains in the volatile data block versions of any modified recoverable parts of the segment. There may be any number of versions due to the nesting of actions and actions sharing the segment. When a top-level action precommits, data must be moved from the volatile data block to the permanent data block prior to shadowing the segment on secondary storage. To simplify the precommit procedure, we allow only one action per segment to pass the precommit point. For example, if actions A and B are both operating on object O and A precommits, B is prevented from precommitting. If B attempts to precommit, the action management system blocks the action. B still may access the object.

During the time precommit and commit are taking place, the virtual memory system must insure that modified pages of the permanent data block remain in memory and undisturbed. The virtual memory system can do this by physically locking the pages in memory, making them read-only. Then the pages can be flushed to disk to build the shadow version of the permanent segment.

The mapping of virtual memory to secondary storage is another of the storage manager's responsibilities. On page faults, the virtual memory system makes use of storage manager calls to locate the backing storage for faulted pages and also to allocate or locate backing storage for virtual pages being paged out.

#### 2.4. Storage Management Implementation

The development of the Clouds storage manager involves the implementation of three components. These are the device object, the partition object, and the segment object. These objects are abstractions of the disk storage available on a Clouds machine. The device object manipulates device storage as a collection of uninterpreted blocks of data, which it transfers in and out of virtual memory. The partition object provides a mechanism for division of device storage for administrative purposes and also is involved in the location of data and the allocation of device storage. The segment object treats device storage as a collection of bytes. In fact, the segment object is just an alternate view of any Clouds object. We consider the device object the lowest level of abstraction and the segment object represents the highest level. In the paragraphs that follow, we describe the current state of the storage manager.

At the device object level, we are developing two disk objects. Clouds disk objects include not only the conventional device driver functions, but also provide necessary support for the recovery mechanisms of the storage manager. Specifically, the Clouds disk objects provide a mechanism, the *flush routine*, which insures that requests scheduled by an action are actually completed before the action commits. This mechanism differs from conventional disk management schemes, where a request may remain enqueued after the process that issued it terminates. The flush routine relies on the *flush table*, a *per device* structure. The table contains an entry for each action; the entry contains a list of requests made by the action and a record of the number of requests pending and completed.

The development of a RL02 disk object has been straightforward and we now have a working version integrated with the Clouds kernel. Minor changes in the way the object formats the medium are anticipated. Additionally, the object must be modified to lock physical pages for I/O transfers, because of the Clouds kernel's use of the virtual memory system as the basic I/O mechanism. The RL02 will allow us to go ahead with the development of kernel and particularly with the testing of the

storage manager. The RL02 will not be the primary disk for the Clouds system, as it holds only 10 Mb on each cartridge. The primary disk for the initial Clouds implementation will be the RA81, a disk object for which is under development in parallel with the development of the RL02 object. Because the RA81 is a "smart" device, progress has been slower and the integration of the facilities required by the Clouds storage manager is more complex. Testing is currently under way on this device. We have kept the device object interface for the two devices uniform and also have attempted to reduce any side-effects so that upon completion of the RA81 object, this object can be use in the place of or along with the RL02.

The next level of abstraction for the storage manager is the partition level, which we have already discussed in some detail. Implementation at this level is just being completed. A partition provides all the structures required to support the creation and management of Clouds objects. Specifically, the partitions provide support for the location of objects and the allocation of disk storage for objects. We have presented the partition structures necessary for these functions, namely the partition directory and record map. There is another structure which, while not necessary for the functionality of the partition object, we believe will considerably improve the efficiency of the partition object's performance.

To understand the significance of this structure note that the Clouds kernel provides for the location-independent invocation of an object operation, which requires the kernel to search for the object at each operation invocation. Object searches are network-wide and several techniques are being developed to short-circuit these searches. One concern is the necessity of going to disk in order to determine if the object is on a partition. Each partition maintains a structure called a *maybe table* which is intended to reduce the number of unnecessary disk accesses during object search (an unnecessary access is one which shows the object is not on the partition in question). The maybe table is a small in-memory representation of the partition membership. A maybe table is an example of a Bloom filter [Bloom70]. The table is a compressed hash table, in which several segment names may hash to the same entry. In trade-off for the reduced size of the table, only a negative response to a query is guaranteed to be correct. Responses indicating the object is on the partition may in fact be wrong and may require the partition object to access the directory on disk. We are really trading accuracy of the responses for speed since in most cases the query can

be answered without an unnecessary access to disk. Because searches are frequent events, we feel that the Maybe table will have a large effect on system performance.

The segment system forms the highest abstraction provided by the storage manager. Disk storage at this level is managed as a collection of arbitrarily sized segments, which generally represent some Clouds object. Segments provide a uniform interface through which the Clouds kernel can manipulate the object. Although segments conceptually provide a simple view of storage as a sequence of bytes, the actual implementation on disk is quite different. A disk segment is a tree structure, which has as its root a *segment header*. This header contains all information pertinent to the segment and is the entry into the segment (an entry in the partition directory points to the segment header). The leaves of the segment tree are the data records on the segment. The internal nodes of the tree consist of link records, providing connectivity between the segment header and the data records.

In addition, the segment system provides the recovery protocols discussed earlier. Implementation of the segment system is currently in progress.

As discussed above, the segment system, the action management system and object management system are involved in the management of virtual memory with respect to the mapping of objects. We are finalizing the extent of each system's responsibilities and influence in the virtual memory and the cooperation needed between the systems. For example, the storage manager shares with the object manager the responsibility for mapping the on-disk version of the segment to the virtual memory version. Each segment has one or more windows which represent chunks of the segments which are actually mapped into virtual memory. This allows the mapping of portions of large objects into virtual memory, avoiding the cost of mapping the entire object. The mapping of each object in memory will consist of several standard windows: a code window for the executable portion of the object; a permanent object data window; and a volatile data window. If the object is recoverable the permanent data window actually consists of several optional windows. There may be a static non-recoverable data window, containing object data not considered necessary as part of the recoverable object state. There may also be a static recoverable data window, which contains (part of) the recoverable object state. Finally, there may be a permanent heap window, which is used in objects which provide customized recovery [Wilkes85a, Wilkes85b]. Each of these windows may be mapped to different

partitions. For instance, the code window and static non-recoverable windows are mapped onto the disk segment image itself. The volatile heap window and recoverable windows are mapped to a paging partition, a special partition reserved simply for providing backing store. In the latter case, the storage manager is really providing two sets of mappings from disk to virtual memory: one for handling page faults and the other for handling the recovery aspects of object management.

The storage manager also makes use of the virtual memory system to assist the action management system in the committing of actions. The segment system uses virtual memory structures to determine which segment pages have been modified and then, based on its own information as to the structure of the segment, decides which segment pages must be shadowed to provide the necessary recovery. Because the storage manager is aware of the segment's virtual memory mapping, and the special attributes of the standard windows, it knows which modified pages actually need to be shadowed and which can be simply written to disk.

A technical report [Pitts85] which summarizes all of the work which has been done on the storage manager was attached to Quarterly Progress Report 4 as Appendix A.

#### 2.4.1. References

[Bloom70] Bloom, B.H., "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, No. 13, Vol. 7 (July 1970), pp.422-426.

[Pitts85] Pitts, D.V. and E.H. Spafford, "Notes on a Storage Manager for the Clouds Kernel" Technical Report GIT-ICS-85/02, School of Information and Computer Science, Georgia Institute of Technology, October 1985.

[Wilkes85a]

Wilkes, C. T., "Programming Methodologies for Resilience and Availability," Ph.D. Thesis Proposal, School of Information and Computer Science, Georgia Institute of Technology, January 1985.

[Wilkes85b]

Wilkes, C. T., "Preliminary Aeolus Reference Manual," Technical Report

### 3. Language Support for Robust Distributed Programs

#### 3.1. The Design of Aeolus

The major design goal of our language Aeolus is to make possible access to the features of the Clouds system from a powerful systems programming language which supplies those features—such as strong typing—which aid in the quick development of error-free programs, yet allows those features to be explicitly circumvented when necessary.

The major structuring features in Aeolus are processes and objects. Objects have two purposes in Aeolus: to provide support for data abstraction, and to reflect the recoverability and synchronization capabilities provided by the Clouds kernel. It has been argued elsewhere [Allc82] that the object construct provides a powerful tool for the organization of programs for recovery, both from the standpoint of the programmer and of the system. Objects may rely on the automatic operating system / run-time system support for synchronization and recovery (*autorecoverable* and *autosynch* objects). Alternatively, using powerful features provided by the language and the Clouds system, the programmer may take advantage of semantic knowledge about the application to explicitly code more appropriate recoverability and synchronization. However, Aeolus objects also provide abstraction features even when synchronization and recovery are not required. These *non-Clouds* objects provide a logical framework for the organization of modules for separate compilation.

##### 3.1.1. Features for Systems Programming

In keeping with its purpose as a systems programming language, Aeolus incorporates several features which give the programmer access to the hardware and the lower levels of the systems software, as well as “convenience” features which allow more efficient coding, including:

- a full range of assignment and bit-manipulation operators similar to those in the C language;

- features for register optimization, such as a special *index* type for loop counters

and array references;

the option of specifying *inline* expansion of a procedure;

a facility for specifying *arbitrary* procedure argument lists of unspecified length and (predefined) types (similar to the *nospread* arglists of Interlisp);

and the ability to specify storage addresses for variables, as well as some capabilities for setting and doing arithmetic on pointers.

However, most of the power of Aeolus as a systems programming language, aside from the access it provides to the features of the Clouds system, lies in the ability it gives the programmer to specify low-level data structures as abstract data types, and in the treatment of the underlying hardware as an object with operations on its state available from the language.

In addition to the usual structured types (records and arrays), Aeolus provides a *structure* type, which allows the programmer to specify abstract types for the manipulation of bitfields. The *structure* is similar to the *packed record* construct of Pascal, except that the programmer indicates that its fields should fit one of the addressable entities defined by the target computer (byte, word, doubleword, quadword, etc.), and this correspondence is checked by the compiler. This provides a secure mechanism allowing bit fields within a low-level data structure to be referenced by name. Aeolus also provides the *byte* and *word* types as predefined objects. These objects have operations permitting manipulations similar to those of the *bitset* type of Modula-2. The programmer may define similar objects for bit strings of other lengths.

The ability to inspect and change the state of the hardware is also important in systems programming. Access to the underlying hardware is provided by the operations of special Aeolus objects. We call such an object a *pseudo-object* since only one instance of it may exist, whereas there may be an arbitrary number of instances of a normal object. An example of a pseudo-object is `PC_System`. This pseudo-object gives access to the registers and ports of a PC's microprocessor, and through the ports to the other system components, such as the interrupt controller, device controllers, and modem registers. For example, the `IN_BYTE` and `OUT_BYTE` operations of `PC_System` allow values to be input and output from the byte ports of a PC; other `PC_System` operations provide such capabilities as access to the register set, flags, and interrupt mechanism. These operations typically compile inline to a single machine instruction. For considerations of efficiency, some operations in hardware pseudo-

objects may give access to special instructions of the target machine, such as the string instructions of the PC or the polynomial instructions of the VAX.

### 3.1.2. Features for Object and Action Programming

The design of Aeolus is intended to support the recovery and synchronization capabilities of the Clouds system in a high-level systems programming language. Objects in Aeolus, besides providing an organizational tool for secure separate compilation, give access to the recovery properties of Clouds objects. Thus, if an Aeolus object is designated as *recoverable* or *autorecoverable*, the Clouds kernel mechanisms are used for invocations of its operations, allowing the system to control the recoverability properties of the object's state. In the remainder of this section, the features provided by Aeolus for accessing these features of Clouds are examined.

The code for an Aeolus object has two parts. The *definition* part is seen both by the object itself when it is being compiled, and by all other objects or programs which use that object. Compilation of a definition part produces a symbol table file which is used for type checking among these separate compilations. It can contain specifications of public types and constants defined by the object, and the interface definitions of the object's operations. Definition parts may not contain variable declarations. The *implementation* part contains the actual code of the operations, along with any needed local (private) type, constant, or procedure definitions. Local variables of an object share the lifetime of the object instance to which they belong, and thus act as "own" variables. This separation of definition and implementation provides a safe separate compilation mechanism similar to *packages* in Ada (TM) or *modules* in Modula-2.

In the header of an object definition, the programmer may specify the object *class* as being *pseudo*, *local*, *nonrecoverable*, *recoverable*, or *autorecoverable*. The classes *pseudo* and *local* are called the *non-Clouds* object classes; the classes *nonrecoverable*, *recoverable*, and *autorecoverable* are called the *Clouds* classes. If the object class is specified as being *pseudo*, the object is treated as being simply a separate compilation module; pseudo-objects are used as a simple separate compilation mechanism, as interfacing to the runtime system and kernel services, and for integrating objects written in other languages into Aeolus systems. *Local* objects provide some of the functionality of Clouds objects (such as access to multiple object instances) without the expense of

the Clouds object management facilities; however, local objects have no existence independent of a process, as Clouds objects may have. The simplest Clouds object class, *nonrecoverable*, makes use of the Clouds object management facilities but does not use the action management. Objects of the *autorecoverable* class, however, provide fully automatic access to the Clouds action management facilities. The entire object state is made recoverable, and default handlers for the *action events* (such as ABORT and COMMIT) are provided by the compiler/runtime system. Thus, the programmer may gain access to the action mechanisms of the Clouds system with a single keyword. However, the full power of the automatic Clouds action mechanisms may be unnecessary and inefficient in some cases. For those cases, the Aeolus/Clouds system provides mechanisms which allow the user to explicitly program recovery strategies tailored to the individual requirements of the problem at hand. If the object is specified as being *recoverable*, the programmer may specify part of the object state as being recoverable and may provide alternate handlers for action events.

The Aeolus language also provides access to the synchronization mechanisms of the Clouds system. When the *autosynch* object attribute is specified in an object definition header, it indicates that the default system synchronization procedures are to be used on the object's operations to provide concurrency atomicity. If the *autosynch* attribute is not specified, synchronization may be explicitly programmed using operations on the *lock* type provided by the language. A Clouds lock [Allc83b] is not associated with a physical object, but rather with values in the domain of the object. Thus—for example—a file name may be locked, even if a physical file with that name does not yet exist.

Object operations are programmed like procedures. An operation invocation looks like a procedure invocation with a prefix indicating the object instance upon which to operate:

<object instance id> @ <operation id> ( <actual param list> )

An object instance may be created by declaring a variable of that object type, and then allocating the instance's data storage on the heap using an extended version of the allocation function, or by associating the variable with a "permanent" object, much as a file variable can be associated with a physical file in Pascal.

An operation or local procedure of a recoverable or autorecoverable Aeolus object may be invoked as an action. The invocation of an action is similar to a procedure or operation invocation; however, a unique *action-id* is created by a Clouds action manager for the invocation, which may be assigned to a variable of the invoking procedure, for example:

```
actionID := action ( object1 @ op1 ( param1, param2 ) )
```

This *action-id* variable may be used to retrieve information from the system about the status of the action, or to abort the action, using calls to a Clouds action manager. This mechanism allows general control structures to be formulated, e.g., for the concurrent invocation of actions.

### 3.2. Programming Methodology

The features of Aeolus described above (and in [Wilk85b]) provide easy access to the synchronization and recovery features of Clouds, and thus they provide a framework within which to study programming methodologies suitable for action-object systems such as Clouds. This study should lead to the design of high-level language features to support that methodology. Thus, our interest in Aeolus lies not so much in the language itself as in studying the sort of programming which may be done with it.

We have found Aeolus to be effective as a systems programming language during our studies of programming systems objects such as communications handlers for the Clouds workstations. In particular, the clarity of interface definitions made possible by use of pseudo-objects is extremely valuable for encapsulation of hardware details in such hardware-dependent programming. Through our experience with developing systems objects, we have come to understand techniques for using subactions as "firewalls" to limit the effect of failures. We have found that Allchin's generalized lock mechanism makes it relatively easy to specify special-purpose synchronization rules dependent on object semantics.

We intend to use Aeolus as a framework within which to study programming methodologies for action-object systems. Among the hardest questions which need more study is how replication can most effectively be used to provide availability.

Actions and resilient objects ensure that failures are not catastrophic, but they are concerned with data integrity, not with how a program reacts to failures. The availability question involves use of multiple objects on different nodes to represent a single resource, thus providing continued access to the resource in the presence of individual node failures. Algorithms for read and write access to such resources must be developed and evaluated. The recent paper by Daniels and Spector [Dani83] is one example of such an algorithm.

We must also consider possible representations of work so that it may be restarted; this is an area that has been until recently unexplored [McKe84]. Most of the work on actions and objects has been oriented toward protection of data from failures. The fact that processes are considered to be an important, independent component supported by the Clouds architecture gives us a point of departure for this study. McKendry's work on Petri nets [McKe84] lays the groundwork for an attack on this problem within the framework of Clouds. If we view a program as a collection of processes interacting through shared objects, some features akin to the process interconnection specifications of Pronet [Macc82] may prove to be useful.

Our initial studies in programming methodologies for resilience and availability are described in [Wilk85a]; there, a plan is presented for determining such methodologies appropriate to the design of objects needed in the Clouds system. Examples of a replicated object exhibiting the properties of resilience and availability are given there, as well as a preliminary design for a *permanent heap*, part of the run-time support necessary for the Aeolus/Clouds system to provide these properties. The issues with which we are concerned include the use of semantic knowledge of objects in the programming of replication; trade-offs between consistency and availability; the appropriateness of current programming models for replicated data; and the support needed from the operating system and language runtime system to ensure availability and forward progress of processes. As we progress with these studies, we will take advantage of our experience in the implementation of the Aeolus runtime system and its interaction with the action and object managers of the Clouds system.

### 3.3. Implementation

A compiler for Aeolus is currently under development on one of the DEC VAX 11/750 computers of the Clouds project under Berkeley Unix (TM) Version 4.2. We

are using the *Amsterdam Compiler Kit* (ACK) [Tane83] to generate code generators for Aeolus for both the Clouds VAXes and the individual work stations which the Clouds system will use to interface to the VAXes. Work on the semantic routines for Aeolus is proceeding in parallel with the development of routines to generate intermediate code for ACK. This work is being done in *Pastel*, an extended Pascal dialect developed at the Lawrence Livermore National Laboratory.

The code-generation work is progressing quite well; we have been able to generate and execute code for object invocations which do not involve the facilities of the Clouds kernel or object managers (that is, code for what we call "non-Clouds objects").

Work is also progressing on the implementation of facilities for generating actual "Clouds objects." This entails the definition of the interface to the Clouds object and action managers, which will serve as an intermediary between user programs and the kernel facilities. Thus, the members of the compiler group are working with members of the kernel group on the definition and implementation of the action and object managers. These interfaces are now well-defined, and we expect the Aeolus compiler to be capable of interfacing with the action and object managers of the Clouds system, and thus to be capable of invocations on actual Clouds objects, by the end of 1985. Actual testing of object and action management calls awaits the implementation of the requisite Clouds system services, which is expected in the first quarter of 1986.

As mentioned above, the design of the interfaces of the runtime system with the Clouds action and object managers is essentially complete. Members of the Aeolus group have been working with members of the kernel group to design these interfaces and strategies for efficient action management. Of particular importance are our designs for support of recoverable areas in Clouds objects; these constructs enable the Aeolus language (in conjunction with the action management system) to provide *view atomicity* in addition to the *failure atomicity* provided by the kernel. Each action which touches an object which has a recoverable area gets its own copy (or *version*) of that recoverable area on which it may make its changes; when a nested action commits, it propagates its version of the recoverable area to its parent. View atomicity ensures that each action in the action tree which accesses an object sees the correct version of the data in that recoverable area. We have developed a technique for implementing recoverable areas using partial replacement of the object page table entries which

provides view atomicity without causing a time penalty for access to the data in the recoverable area. Rather, a small penalty is paid upon process exchange if a process is associated with an action.

### 3.4. Design Refinements

The work of the Aeolus group during recent months has been concentrated on the refinement and rationalization of the language design. The design of the language has undergone a significant reworking, especially those parts of the design concerned with specification of types. In view of one of the Aeolus design goals of providing the power necessary for systems programming without sacrificing the advantages of strong type checking, we wished to provide dynamic (flexible) data types; however, we felt that our previous design for this violated the goals of simplicity and readability. Our reworked design integrates flexible types into the language in a much cleaner manner, within the framework of general parameterized types. The changes to the design have been incorporated into the language description [Wilk85b], along with the interfaces to the Clouds system object and action managers. The revised language definition was attached to Quarterly Progress Report 4 as Appendix B.

These design changes have now been incorporated into the symbol table of the compiler and new semantic routines necessitated by the changes are being implemented. We have also taken advantage of the redesign to streamline parts of the semantic routine structure, taking into account our previous implementation experience. Work on the implementation is accelerating now that these changes are understood.

### 3.5. References

- [Dani83] Daniels, D., and A. Z. Spector, "An Algorithm for Replicated Directories," *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, August 1983.
- [Macc82] Maccabe, A. B., and R. J. LeBlanc, "The Design of a Programming Language Based On Connectivity Networks," *Proceedings of the Third International Conference on Distributed Computing Systems*, Miami / Fort Lauderdale, October 1982.

- [McKe84] McKendry, M. S., "Ordering Actions for Visibility," *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, Silver Spring, Maryland, October 1984 (also available as Technical Report GIT-ICS-84/05).
- [Tane83] Tanenbaum, A. S., H. van Staveren, E. G. Keizer, and J. W. Stevenson, "A Practical Tool Kit for Making Portable Compilers," *Communications of the ACM* 26, 9, September 1983.
- [Wilk85a] Wilkes, C. T., "Programming Methodologies for Resilience and Availability," Ph.D. Thesis Proposal, School of Information and Computer Science, Georgia Institute of Technology, January 1985.
- [Wilk85b] Wilkes, C. T., "Preliminary Aeolus Reference Manual," Technical Report GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, July 1985 (revised October 1985).

#### 4. Conclusions

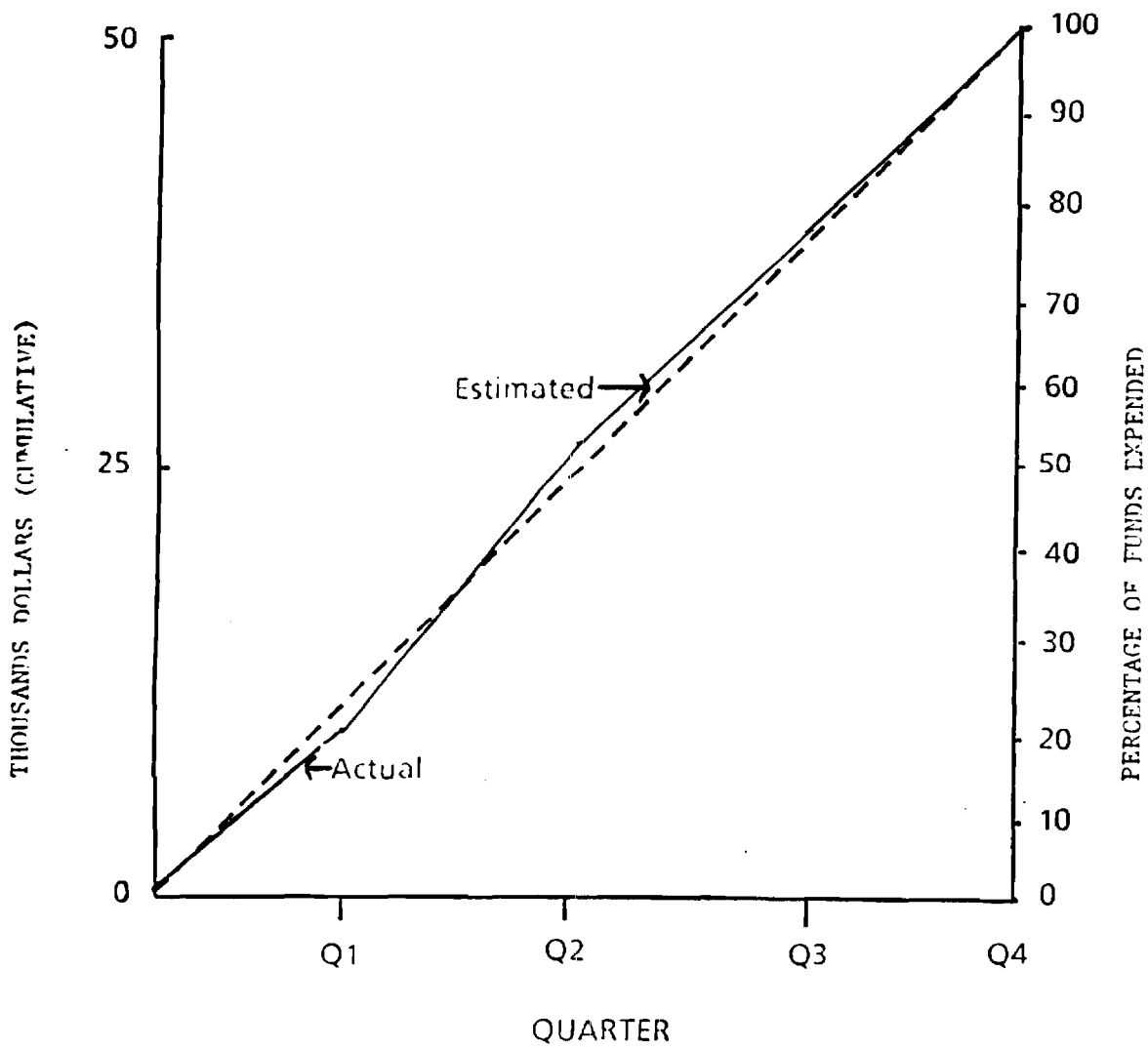
Work on both of the two tasks of the project has been very productive over the last year. In conjunction with related work on the Clouds kernel and the action management system, we anticipate having a working Clouds prototype by early in 1986. The storage manager designed and currently being implemented under the Distributed File Systems task obviously is strongly related to the kernel and thus must be combined with the kernel for testing. That integration should take place in the immediate future. The next phase of this work, which includes performance measurement and analysis followed by design refinement, will then be possible.

Under the Language Support for Robust Distributed Computing task, we have produced a language (Aeolus) which includes features that match the capabilities for action and object management provided by the Clouds kernel. This language has enabled us to begin our studies of programming methodologies for action/object programming. Further study in this area will be a major ongoing research effort. We have also made substantial progress on a compiler for Aeolus. It will be ready for use by the time the kernel and storage manager are integrated and available for managing Aeolus objects and processes. The compiler interface to the action management system has been defined and the compiler should be capable of generating code to utilize

this interface even prior to the implementation of the action manager.

In summary, our efforts under this project have been instrumental in the development of a prototype of the Clouds reliable distributed computing system concept. Work to be done in the next year will provide significant evidence of the viability of the Clouds concept.

# FUNDS EXPENDITURE GRAPH



FUNDS EXPENDITURE REPORT

Column A -----	Column B -----	Column C -----	Column D -----	Column E -----	Column F -----
ORIGINAL PROPOSAL	Latest Accepted Revised Proposal	Reporting Quarter Expenditures	Cumulative Expenditures to Date Total Man Hours Dollar Value Pct. Dollar Value	Cost to Complete Estimate	Latest Cost Estimate
Direct Labor					
Type	Number of Hours	Hourly Rate	Dollar Total	Number of Hours	Dollar Total
-----	-----	-----	-----	-----	-----
DI	350	23.77	\$8320.00	430	\$10220.96
BRA	1300	11.41	\$14833.00	1648	\$18802.35
Clerical	175	6.74	\$1180.00	0	\$0.00
			-----	-----	-----
Total Direct Labor			\$24333.00		\$29023.31
Burden @ 24.6%			\$2337.00		\$2387.93
(21.0% starting 7/1/85)					
(Excluding BRA Labor)					
			-----	-----	-----
Total Direct Labor and Burden			\$26670.00		\$31411.24
TRAVEL EXPENSE			\$2500.00		\$1193.74
GENERAL & ADMINISTRATIVE EXPENSE			\$1500.00		\$413.46
COMPUTING CHARGES			\$1500.00		\$1565.18
			-----	-----	-----
TOTAL DIRECT COSTS			\$32170.00		\$34583.62
DIRECT COSTS @ 55.3%			\$17790.00		\$19959.38
(63.5% starting 7/1/85)					
			-----	-----	-----
CONTRACT PRICE			\$49960.00		\$54543.00
COMMITMENTS AND EXPENDITURES				\$17144.20	\$54543.00

*Programming Methodologies for Resilience and Availability*

C. Thomas Wilkes

School of Information and Computer Science  
Georgia Institute of Technology  
Atlanta, GA 30332

## ABSTRACT

The goal of the *Clouds* project at Georgia Tech is the implementation of a fault-tolerant distributed operating system based on the notions of *objects* and *actions*, which will provide an environment for the construction of reliable applications. As part of the *Clouds* project, we are designing and implementing a high-level language in which those levels of the *Clouds* system above the kernel level will be implemented. The *Aeolus* language provides access to the synchronization and recovery features of *Clouds*. It also provides a framework with which to study programming methodologies suitable for action-object systems such as *Clouds*.

This proposal describes a plan for determining programming methodologies appropriate to the design of objects needed in the *Clouds* system. Among the properties needed by these objects are *resilience* and *availability*. Examples of a replicated object exhibiting these properties are given, as well as a preliminary design for a *permanent heap*, part of the run-time support necessary for the *Aeolus/Clouds* system to provide these properties.

## 1. INTRODUCTION

Among the benefits promised by advocates of distributed computing are improvements in system fault tolerance and reliability, increased availability of data and services, and faster response through use of distributed programs. Interest in reliability has grown as distributed systems have come to be used in an ever widening set of applications, including critical control systems. In the past, fault tolerance has principally been the concern of hardware designers, who mainly used redundancy as a solution. More recently, it has been realized that maintaining the integrity of distributed data is a crucial concern in providing the benefits listed above. Accordingly, there has been a growth in research interest in techniques for providing the required data integrity in the presence of hardware failures and concurrently executing processes.

The *Clouds* project at Georgia Tech [Allc82, Allc83a, Allc83b] is one of a number of recent proposals in which reliability in a distributed system is based on the use of *atomic actions*, a generalization of the transaction concept of distributed databases. As part of the *Clouds* project, we are designing and implementing a high-level language which will provide access to the synchronization and recovery features of the *Clouds* system; this language will be used to implement those levels of the *Clouds* system above the kernel level. It will also provide a framework within which to study programming methodologies suitable for systems based on the action concept, such as *Clouds*. Among the properties needed by systems data structures, the design of which must be addressed by such methodologies, are *resilience* -- survivability and consistency of the data despite crashes and other faults -- and *availability* -- increased possibility of access to replicated data despite network partitions or failures of some sites in a multicomputer system.

This proposal describes a plan for studying such methodologies, in particular those needed in the design of the type of data structures necessary for the implementation of the *Clouds* system. Section 2 of this proposal presents the problem explored by this work and describes the environment in which it will be examined (the *Clouds* system). Section 3 describes the plan of the research to be performed, and presents examples of a replicated object exhibiting the properties of *resilience* and *availability*, as well as a preliminary design for a *permanent heap*, part of the run-time support necessary for the *Aeolus/Clouds* system to provide these properties. An outline of the proposed dissertation is presented in Section 4.

## 2. PROBLEM STATEMENT AND BACKGROUND

The purpose of the proposed research is to explore programming methodologies for action-based systems appropriate to the design of data structures exhibiting the properties of resilience and availability. In this section, the environment in which this study will be carried out is described, and a rationale for the need of such programming methodologies is presented.

### 2.1. The Clouds System

The goal of the Clouds project at Georgia Tech is to allow the construction of reliable application systems on unreliable hardware. We use the notion of an *object* to represent system components, such as directories or queues. An object consists of data and a set of operations on that data which maintain a set of associated invariants. A set of changes to objects is grouped into an action, a unit of work which appears to be *atomic* to its environment (in particular, to other actions). Objects are passive in the Clouds architecture; thus, *processes*, which may represent a single top-level or nested action, are used to provide activity in the system.

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

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

The mechanisms developed for the support of transactions in database systems, as well as the traditional operating system synchronization mechanisms, have been found to be insufficient for the support of the action-object approach in operating systems. In particular, the problems of ordering and atomicity for nested actions, and several simplifications which apply to many operating systems problems, are discussed in [McKe84b]. In particular, it is shown that through the use of *per-action variables*, it is not necessary to maintain complete versions of recoverable data for many types of systems data structures, giving substantial gains in space and time performance. The expediciencies made possible by these simplifications make the use of the action-object approach in the Clouds system viable.

These mechanisms provided by the Clouds architecture are used to support the operating system itself and its services. Thus, the system itself is decentralized and resilient. The Clouds system may be considered to consist of a set of fault-tolerant objects (*servers*) which in combination provide a reliable environment for applications.

### 2.2. The Need for an Action-Based Programming Methodology

Actions are the key feature for guaranteeing data consistency. The "all-or-nothing" nature of actions really solves two problems. When an action fails, its effects are automatically undone; so, actions which fail due to machine failures cannot leave objects in an inconsistent state. Additionally, the required serializability of actions provides a coarse-grained synchronization among them. (Other features may be used to provide more concurrency by supporting synchronization at a lower level.) Actions which are aborted for logical reasons (e.g., deadlock) again can have no visible effects on the state of any object. Thus the action concept successfully broadens the recovery viewpoint provided by checkpoints, since it encompasses all the changes to any number of objects made by an arbitrarily complex action.

Actions alone do not provide all of the generally desired capabilities, since they do not address the question of the resiliency of individual objects. That is, they do not contribute toward the recovery of objects

located on machines that fail. Rather, they guarantee the integrity of surviving objects. Both Argus and Clouds support resilience through use of *stable storage*. (Stable storage has the property that information entrusted to it is extremely unlikely to be lost.) Various features are provided which cause the object support system to record sufficient information on stable storage so that the state of an object (*guardian* in Argus) may be recovered after a hardware failure. Note that for a combination of consistency and resilience, the state of an object must be written to stable storage whenever an action which modified the object commits (presuming that pessimistic recovery is being used).

Writing the state on an object to stable storage is, of course, just checkpointing. It is the coupling with the action mechanism which makes checkpointing of objects effective. That is, part of the implementation of a commit is a checkpoint of all affected objects. Thus checkpointing is made an effective means for providing consistent, resilient objects.

The mechanism for specifying just what must be written is one way in which Argus, Clouds and other proposals differ. In Argus, all *mutex* objects within a guardian are written. As suggested by the name, *mutex* objects also have certain synchronization properties, relating to their accessibility to concurrently executing actions. Clouds, on the other hand, allows an entire object or any data object within it to be specified as *recoverable*. As would be expected, if the entire object is recoverable, then all of its contained data objects are written to stable storage when a relevant commit occurs. Both of these approaches exemplify implicit specifications of what must be saved for recovery purposes. Yet another approach would be to require the programmer who defines an object to provide an explicit *write-to-stable-storage* operation to be invoked by the object management system at appropriate times. This variety of proposals reflects the need for study of a programming methodology based on use of objects and actions, so that we can determine just what kinds of features are most effective.

The Clouds architecture goes beyond others in that it can support actions that involve objects on more than one machine. In other words, a remote procedure call can be done without creation of a nested action. Allchin's work [Allc83b] provides a definition of the basic capabilities supported by the Clouds architecture and a design for their implementation. Now that that implementation is in progress, we are studying how these capabilities may be applied. In particular, we wish to study a programming methodology for systems like Clouds. The semantic knowledge about objects afforded by the encapsulation in object-oriented systems like Clouds should provide opportunities for optimizations in the treatment of the synchronization and recovery of replicated data unavailable in the simple read-write database model.

As part of the Clouds project, we are designing and implementing high-level systems programming language called *Aeolus* (after the king of the winds in Greek mythology). An overview of the *Aeolus* language is presented in [LeB185] (most of the materials in this and the preceding sections are also derived from this paper). *Aeolus* gives the programmer access to the features of the Clouds system discussed above. However, we also intend to use *Aeolus* as a framework for studying the sort of programming methodology appropriate to Clouds. This study should lead to the design of high-level language features to support that methodology. Additionally, it should suggest what capabilities are desirable both in the *Aeolus* run-time system and in the underlying action and object management support of the Clouds system.

### 3. THE PROPOSED RESEARCH

In this section, a plan for the proposed research is presented, as well as some preliminary examples of the sort of studies which will be made.

#### 3.1. Plan of Research

The investigation of programming methodologies appropriate to systems objects having the properties of resilience and availability necessitates the programming of such objects. In the course of our studies, we propose to design simple versions of fault-tolerant servers which will be useful in the Clouds system, such as name servers, directory managers, system queues, etc. These designs may not be complete, since the aim is to study the issues of availability and resilience in terms of these data structures, rather than such issues as

naming in distributed systems. Complete designs of these objects will be left for those researchers studying the relevant issues within the environment of the Clouds system.

Among the issues which will be treated during this investigation are:

*Use of semantic knowledge of objects in programming of replication*

Can significant savings be achieved in programming the synchronization and recovery of replicated data by taking advantage of semantic knowledge of complex objects rather than using a simple read-write model?

*Trade-offs between consistency and availability*

Is it necessary, in programming replicated system services, to maintain strict consistency (serializability) among replicates in all cases, or are there cases in which global consistency may be weakened in favor of availability of services?

*Blocking (pessimistic) vs. non-blocking (optimistic) replication methods*

Again, there are trade-offs here between consistency and availability; for the systems objects of interest here, is the increased availability during partitions or site failures afforded by optimistic methods worth the possible cost incurred by the necessity to resolve inconsistencies introduced among the replicates during the partition?

*Appropriateness of current programming models for replicated data*

Are programming models useful in programming non-replicated objects also useful when replication is to be taken into account? That is, do the present models suffice, or must new models be developed for efficient programming of replicated objects?

*Support needed for replication*

What support is needed from the run-time system and from the operating system to support the programming of replicated objects?

*Support needed to ensure forward progress*

What support is needed from the job scheduler [McKe84a] to continue execution of operations on replicated objects despite failures?

Based on the knowledge gleaned about programming for resilience and availability during these studies, we propose to design language features to aid in programming fault-tolerant, replicated objects. Besides this upwards migration of knowledge, a downwards migration -- capabilities required of the language run-time support, as well as the action and object managers -- should manifest itself.

### 3.2. Preliminary Studies

As examples of the types of explorations which will be made in the course of this research, some preliminary studies are presented here. These examples include investigation of the programming of replicated objects using *recoverable variables* and using *per-action* and *permanent variables*. Also, a tentative design for a permanent heap to provide part of the run-time support for these objects is presented.

#### 3.2.1. Replication Using Recoverable Variables

An example of an Aeolus object programmed to provide availability via replication is shown in Example 1. This object, which implements a simple symbol table, is derived from a single-copy (non-replicated) object (discussed in [LeB185]) which uses actions to provide recovery "firewalls" around its critical operations and the Clouds lock mechanism to specify synchronization rules which allow a high degree of concurrency the use of its operations.

The replicated SYMTAB object shown here uses *recoverable variables* to provide resilience of data in the face of failures. Recoverable variables, discussed in [Allc83b], are similar to *versions* of distributed database work, and require the creation of a new version (copy) of a recoverable variable for each action which modifies the variable. These versions are maintained on a *version stack* by the action managers of the Clouds system, which control visibility of the versions to nodes of the action tree, as well as writing versions to stable storage upon action commit.

Availability of the SYMTAB object is achieved by replication. Two instances of this object (*partners*) are created by the parent process, one of which is arbitrarily made the *master* instance, while the other is made the *slave*. Operations may be invoked on either of these instances; however, the slave instance merely relays the operation requests to the master. The master instance will then carry out the operations both locally and at the slave instance (by means of special operations) as if they had been originally invoked at the master. Were the instances equal partners (that is, if operations could be carried out at either instance), it would be necessary for each instance to obtain appropriate locks both locally and at the partner in order to maintain consistency, which could lead to deadlock unless some sort of global locking mechanism is available. In our implementation, locks need be obtained only at the master; thus, deadlock problems are avoided without the need for global locks. Should the slave instance become unavailable to the master (because of network partition or failure of the slave's site), the master will no longer attempt to replicate operations at the slave until a reinitialization operation is invoked by the Clouds action/object management support. Should the master instance become unavailable to the slave, the slave makes itself the master and carries out operations locally until reinitialization is invoked. When reinitialization occurs, the original master instance is again made master, and the states of the two instances are merged. The merge process is aided by maintenance of a *version vector*, which contains a sort of timestamp of both the latest version of the data maintained locally and the last version which was known to be consistent with the partner instance.

The implementation of the replicated SYMTAB object stresses availability at the expense of strict consistency among the replicated instances across partitions by using an optimistic (non-blocking) recovery method (see the section on "Related Work"). Rather, it seeks to maintain a "reasonable" view of the data at each instance, and resolves inconsistencies during the state-merge process using backout or compensatory operations. The programming of the merge process takes advantage of semantic knowledge of the object. Thus, the replicated SYMTAB object may be used -- despite site failures or partitions -- at any node at which an instance is available, without running the risk of an operation being blocked in order to maintain global consistency.

*Example 1. Replicated SYMTAB object using recoverable variables*

```
implementation of object symtab ( replicate_number : integer ) is
    ! Two-copy master/slave symbol table object using the action management
    ! facilities of Aeolus/Clouds for recovery firewalls and the lock
    ! mechanisms for synchronization, and demonstrating optimistic
    ! (non-blocking) site crash and partition recovery methods.

    ! The definitions of MAXREPLICATE, REPLICATE_RANGE, and VERSION_VECTOR
    ! actually appear in the definition part of SYMTAB, but are shown here for
    ! convenience.

    MAXREPLICATE : const integer := 2

    type replicate_range is 1 .. MAXREPLICATE

    type version_vector is array [replicate_range] of integer

    !
    ! The actual declarations of the implementation part.
    !

    here, there : recoverable replicate_range
                  ! for storing values of replicate numbers

    MAXBUCKET : const integer := 101 ! or whatever

    type hash_range is 1 .. MAXBUCKET

    type ptr_entry is -> symtable_entry

    type symtable_entry is ! just something for demo purposes
        record
            name : valstring ,
            next : ptr_entry
        end record

    type symtab_type is array [hash_range] of ptr_entry

    symtable : recoverable symtab_type

    symentry_lock : lock ( write : [],
                           read : [read] ) domain is hash_range
    ! The SYMENTRY lock allows locking of individual hash buckets in the
    ! symbol table. Several READ operations are allowed to proceed
    ! concurrently, but a WRITE operation blocks all other operations.

    symtable_lock : lock ( write : [write] ,
                           read : [read] )
    ! The SYMTABLE lock allows the entire symbol table to be locked.
    ! This lock is set 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.

    partner : recoverable symtab ! Object pointer to the partner object

    master : boolean ! remember whether this instance is master or slave

    local_version : recoverable version_vector
    ! The LOCAL_VERSION vector is used to store version numbers of the local
    ! state (the HERE entry) and of the last version of the local state
    ! known to be consistent with the partner's state (the THERE entry).
    ! Note, however, that only one copy (per instance) of the actual state
    ! is maintained.
```

```

procedure hash ( name : valstring ) : hash_range is
  ! This HASH function is a local (nonpublic) procedure of
  ! the SYMTAB object.
  begin
    ! the usual type of stuff
  end procedure ! hash !

```

```

procedure send_state () is action
  ! The SEND_STATE operation is called by the partner to copy the local
  ! state to the partner.
  ! Here, knowledge of object semantics can be of great help in fine-tuning
  ! the state transfer process. For example, if the number of changes
  ! expected over an average expected partition period is small with respect
  ! to the total number of symbol table entries, it might be better to keep a
  ! log of those changes made during the partition period and then execute
  ! those changes on the partner's state. The method shown below for copying
  ! the entire local state to the partner assumes that the number of changes
  ! will be large with respect to total size. Also, the method shown --
  ! total reconstruction of the symbol table -- is used
  ! since the symbol table data structure uses physical pointers. Had the
  ! data structure used integer indices instead, the state could have been
  ! copied directly to the partner.

```

```

next_entry : ptr_entry
i          : index hash_range

begin
  for i := 1 to MAXBUCKET loop
    next_entry := symtable [i]
    while next_entry <> NIL loop
      partner @ sym_insert (next_entry -> .name, local_version [here])
      next_entry := next_entry -> .next
    end loop
  end loop
  local_version [there] := local_version [here]
end procedure ! send_state !

```

```

procedure receive_state () is action
  ! The RECEIVE_STATE operation is called by the partner's SEND_STATE
  ! operation to receive a copy of the partner's state.
  ! This operation doesn't do anything because of the symbol table data
  ! structure and method of state transfer used above. If the data structure
  ! used indices instead of pointers, this operation would install the copy of
  ! the state sent by the partner.
  begin
    null
  end procedure ! receive_state !

```

```

procedure transfer_state (! partner_version : version_vector ! ) is action
  ! The TRANSFER_STATE operation is called by the MERGE_STATE operation to
  ! have the partner transfer its state to us. If the partner becomes
  ! unavailable, make this instance the master.
  begin
    aid := action (partner @ send_state ())
    if Committed (aid) then
      local_version [here], local_version [there] := partner_version [there]
    else
      partner_available := FALSE
      master             := TRUE
    end if
  end procedure ! transfer_state !

```

```

procedure reconcile_states ( partner_version : version_vector ) is action
! The RECONCILE_STATE procedure is called locally (by the MERGE_STATE
! operation) to merge the local state and the partner's state in
! the case where both states have been updated since a partition
! occurred.
begin
    null ! later
end procedure ! reconcile_states !

```

```

procedure sym_insert (! newname : valstring, newversion : integer !) is action
! The SYM_INSERT operation may be invoked as an action.
! It is called either by the INSERT operation below (if this instance is
! the master), or by the partner as an update operation (if this instance
! is the slave).
! Locks on the symbol table and the particular hash entry concerned are
! obtained by the caller.
! If NEWVERSION is greater than 0 (that is, the SYM_INSERT operation
! was called remotely), then this new version number is installed in the
! LOCAL_VERSION array.

```

```

entry      : ptr_entry
bucket_num : hash_range

```

```

begin
    bucket_num := hash (newname)
    new (entry)
    using ent := entry -> do
        ent.name := newname
        ent.next := symtable [bucket_num] -> .next
    end using
    region symtable [bucket_num] do
        symtable [bucket_num] := entry
    end region
    if newversion > 0 then
        local_version [here], local_version [there] := newversion
    end if
end procedure ! sym_insert !

```

```

procedure insert (! newname : valstring !) is action
! The INSERT operation may be called as an action.
! If this instance is the master, it sets write locks on the symbol table
! and on the hash entry to be changed, and then calls the SYM_INSERT
! operation both locally and at the slave partner (if available) to do
! the actual insertion.
! If this instance is the slave, it calls the master (if available) to do
! the insertion just as if the call had originated there. If the master
! is not available, the slave makes itself the master and performs the
! insertion.
! Note that, since only the master can call the SYM_INSERT operation
! (which actually performs the insertion), it is not necessary to set
! locks at the slave.

newversion : integer
aid        : action_id

begin
  if master then
    newversion := local_version [here] + 1
    SetLock (symtable_lock, write)
    SetLock (symentry_lock, write, hash (newname))
    sym_insert (newname, 0)
    local_version [here] := newversion
    if partner_available then
      aid := action (partner @ sym_insert (newname, newversion))
      Await (aid) ! block until the nested action commits or aborts
      if Committed (aid) then
        local_version [there] := newversion
      else
        partner_available := FALSE
      end if
    end if
  else ! this instance is the slave
    aid := action (partner @ insert (newname))
    Await (aid)
    if not Committed (aid) then ! become master
      partner_available := FALSE
      master := TRUE
      newversion := local_version [here] + 1
      SetLock (symtable_lock, write)
      SetLock (symentry_lock, write, hash (newname))
      sym_insert (newname, 0)
      local_version [here] := newversion
    end if
  end if
end procedure ! insert !

.....

merge_lock : lock ( busy : [] )

```

```

procedure merge_state (! partner_version : version_vector !) is action
! The MERGE_STATE operation is invoked by the master partner after it{
! receives a REINIT invocation after a partition or crash.
! This operation compares the local version vector (LOCAL_VERSION)
! with that of the partner (PARTNER_VERSION), and takes appropriate action
! to merge its state with that of the partner into a single,
! consistent state.
begin
  SetLock (merge_lock, busy)
  master := FALSE
  if local_version [here] = partner_version [there] then
    ! The local and partner states are already consistent
    return .
  elsif local_version [here] = partner_version [here] then
    ! The local state hasn't changed since the last time the
    ! states were consistent; copy the partner's state here
    transfer_state (partner_version)
  elsif local_version [there] = partner_version [there] then
    ! The partner's state has not changed since the last time the
    ! states were consistent, so just copy over the local state
    ! to the partner
    partner @ transfer_state (local_version)
  else
    ! Both the local and the partner's state have changed since
    ! the two states were last consistent (partition case),
    ! so we must merge them
    reconcile_states (partner_version)
  end if
end procedure ! merge_state !

```

```

procedure reinit () is action
! The REINIT operation is invoked by the object manager in charge of this
! instance when the site on which it is running comes back up after a
! crash, or when a partition ends and this instance's state must be merged
! with its partner's state.
! If the MERGE lock is set, the partner has already initiated the merge
! process, and this instance is made the slave.
! If the MERGE lock is not set, this instance is the first to have the
! REINIT operation invoked by the object manager, so it becomes the master,
! invokes the partner's MERGE_STATE operation and passes it the local
! LOCAL_VERSION array; the partner (if available) then determines what
! needs to be done to merge the two states into a consistent state.

```

```

aid : action_id
begin
  if TestLock (merge_lock, busy) then
    return .
  end if
  SetLock (merge_lock, busy)
  master := here = 1 ! arbitrary choice
  if master then
    aid := action (partner @ merge_state (local_version))
    Await (aid)
    partner_available := Committed (aid)
  else
    aid := action (partner @ reinit ())
    Await (aid)
    if not Committed (aid) then
      master := TRUE
      partner_available := FALSE
    end if
  end if
end procedure ! reinit !

```

```
procedure set_partner (! p : symtab, rep_number : replicate_range !) is
  ! The SET_PARTNER operation is used by the creating process to
  ! initialize the PARTNER object pointer.
  begin
    partner := p
    there   := rep_number
  end procedure ! set_partner !

begin ! initialization section
  here      := replicate_number
  master    := here = 1 ! arbitrary choice
  local_version := version_vector[ 0 : MAXREPLICATE ]
  symtab    := symtab_type[ NIL : MAXBUCKET ]
             ! symbol table is initially empty
end implementation.
```

### 3.2.2. Replication Using Permanent and Per-Action Variables

Since the use of a *recoverable* symbol table data structure requires the creation of a complete copy of the symbol table on the version stack for each action which modifies the data structure, the implementation of the replicated SYMTAB object presented in the preceding subsection can become inefficient as the size of the symbol table increases. Fortunately, we can use semantic knowledge about the object to simulate the effect of recoverable variables at a fraction of their cost. The technique which we will use is introduced in [McKe84b].

An implementation of the SYMTAB object which uses this new technique is shown in Example 2. Rather than require the system to maintain a version of the symbol table per action, we will maintain lists of those elements inserted or deleted by each action. These lists are maintained in *per-action variables*, copies of which are created for each action during its BOA (beginning of action) phase; an action may access not only its own per-action variables, but those of its parent (if any). We provide handlers for the *abort* and *nested commit* events of actions, which clean up after action aborts and propagate the values of the per-action variables to the parent of the current action. Only one copy of the entire symbol table data structure is maintained, as a *permanent variable*. Permanent variables are maintained in a special area of per-object storage, and are managed by a shadowing mechanism provided by the Clouds kernel storage management system [Pitt84]. The shadows are created during the *precommit* action event; thus, we provide a handler for the *oplevel precommit* event which performs the actual insertions and deletions on the permanent symbol table, using the INSERTED and DELETED lists which we have propagated up the action tree.

As in Example 1, only the INSERT operation is shown for sake of brevity; however, the form of the DELETE operation would be very similar to that of INSERT, except that items would be added to the DELETED per-action list rather than to the INSERTED list. The processing of the DELETED list during action events is shown in the code for the alternate event handlers.

Example 2. Replicated SYMTAB object using permanent and per-action variables

implementation of object symtab ( replicate\_number : integer ) is

```
! Two-copy master/slave symbol table object using the action management
! facilities of Aeolus/Clouds for recovery firewalls and the lock
! mechanisms for synchronization, and demonstrating optimistic
! (non-blocking) site crash and partition recovery methods.
! Permanent variables, rather than recoverable variables, are used for
! for efficiency.
```

```
! Names are given here for alternate handlers provided for some of the action
! events.
```

action events

```
    abort is sym_abort,
    nested_commit is sym_nested_commit,
    toplevel_precommit is sym_top_precommit
```

```
! The definitions of MAXREPLICATE, REPLICATE_RANGE, and VERSION_VECTOR
! actually appear in the definition part of SYMTAB, but are shown here for
! convenience.
```

```
MAXREPLICATE : const integer := 2
```

```
type replicate_range is 1 .. MAXREPLICATE
```

```
type version_vector is array [replicate_range] of integer
```

```
!
! The actual declarations of the implementation part.
!
```

```
here, there : permanent replicate_range
! for storing values of replicate numbers
```

```
MAXBUCKET : const integer := 101 ! or whatever
```

```
type hash_range is 1 .. MAXBUCKET
```

```
type ptr_entry is -> permanent symtable_entry ! allocate in perm. heap
```

```
type symtable_entry is ! just something for demo purposes
record
    name : valstring ,
    next : ptr_entry
end record
```

```
type symtab_type is permanent array [hash_range] of ptr_entry
! the array of pointers is in perm. storage
```

```
symtable : symtab_type
```

```
symentry_lock : lock ( write : [],
    read : [read] ) domain is hash_range
! The SYMENTRY lock allows locking of individual hash buckets in the
! symbol table. Several READ operations are allowed to proceed
! concurrently, but a WRITE operation blocks all other operations.
```

```
symtable_lock : lock ( write : [write] ,
    read : [read] )
! The SYMTABLE lock allows the entire symbol table to be locked.
! This lock is set 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.
```

```

partner      : permanent symtab  ! Object pointer to the partner object
master       : boolean  ! remember whether this instance is master or slave

local_version : recoverable version_vector
  ! The LOCAL_VERSION vector is used to store version numbers of the local
  ! state (the HERE entry) and of the last version of the local state
  ! known to be consistent with the partner's state (the THERE entry).
  ! Note, however, that only one copy (per instance) of the actual state
  ! is maintained.

! The per-action variables of the SYMTAB object.
! We will maintain lists of those entries inserted and deleted by
! each action. The per-action variables are headers of those lists,
! which are pointers to entries allocated in the permanent heap.
! There are two standard names, Self and Parent, which refer to the
! per-action records of the current action and its parent,
! respectively.
! The PER-ACTION declaration causes a record type with name PERACTION,
! as well as the names Self and Parent with that record type, to be added
! to the Aeolus compiler's symbol table.

per-action is
  record
    inserted, deleted : ptr_entry
  end record
  init peraction"[ NIL:2 ]  ! For initialization of Self at action start

procedure hash ( name : valstring ) : hash_range is
  ! Same as in Example 1.

procedure send_state ( ) is action
  ! Same as in Example 1.

procedure receive_state ( ) is action
  ! Same as in Example 1.

procedure transfer_state (! partner_version : version_vector ! ) is action
  ! Same as in Example 1.

procedure reconcile_states ( partner_version : version_vector ) is action
  ! Same as in Example 1.

```

```

procedure sym_insert (! newname : valstring, newversion : integer !) is action
! The SYM_INSERT operation may be invoked as an action.
! It is called either by the INSERT operation below (if this instance is
! the master), or by the partner as an update operation (if this instance
! is the slave).
! Locks on the symbol table and the particular hash entry concerned are
! obtained by the caller.
! If NEWVERSION is greater than 0 (that is, the SYM_INSERT operation
! was called remotely), then this new version number is installed in the
! LOCAL_VERSION array.
! The insertion is noted on the INSERTED per-action list of the current
! action, but is not actually performed until toplevel precommit.

```

```

entry      : ptr_entry
bucket_num : hash_range

```

```

begin
  bucket_num := hash (newname)
  new (entry)
  using ent := entry -> do
    ent.name := newname
    ent.next := Self.inserted -> .next
  end using
  Self.inserted := entry
  if newversion > 0 then
    local_version [here], local_version [there] := newversion
  end if
end procedure ! sym_insert !

```

```

procedure insert (! newname : valstring !) is action
! Same as in Example 1.
! Note that manipulation of the INSERTED per-action list is performed by
! the SYM_INSERT operation, which is called by the INSERT operation of the
! master.

```

.....

```

procedure sym_abort () is
! The SYM_ABORT procedure is the alternate handler for the ABORT action
! event for the SYMTAB object.
! It frees all space which was allocated in the permanent heap for
! items inserted by the action being aborted.

```

```

entry, next_entry : ptr_entry

```

```

begin
  entry := Self.inserted
  while entry <> NIL loop
    next_entry := entry -> .next
    dispose (entry)
    entry := next_entry
  end loop
end procedure ! sym_abort !

```

```

procedure sym_nested_commit () is
! The SYM_NESTED_COMMIT procedure is the alternate handler for the
! NESTED-COMMIT action event for the SYMTAB object.
! It adds the INSERTED and DELETED lists for the nested action being
! committed to the beginning of the respective lists of the action's parent.

entry : ptr_entry

begin
entry := Self.inserted
if entry <> NIL then
loop      ! find the end of the INSERTED list
  if entry -> .next = NIL then
    exit .
  end if
  entry := entry -> .next
end loop
entry -> .next := Parent.inserted
Parent.inserted := entry
end if

entry := Self.deleted
if entry <> NIL then
loop      ! find the end of the DELETED list
  if entry -> .next = NIL then
    exit .
  end if
  entry := entry -> .next
end loop
entry -> .next := Parent.deleted
Parent.deleted := entry
end if
end procedure ! sym_nested_commit !

```

```

procedure sym_top_precommit () is
! The SYM_TOP_PRECOMMIT procedure is the alternate handler for the
! TOPLEVEL-PRECOMMIT action event for the SYMTAB object.
! It inserts or deletes each item in the INSERTED or DELETED list for this
! action into (or out of) the permanent symbol table. At this point,
! the memory management system will create shadow versions of the pages
! in the permanent version affected by these changes.
! Note that, since the action management system promises that only one
! action can enter its PRECOMMIT stage at a time, no further locking for
! mutual exclusion is necessary.

entry, next_entry : ptr_entry
place, prev_place : ptr_entry
bucket_num       : hash_range

begin
  entry := Self.inserted
  while entry <> NIL loop
    next_entry := entry -> .next
    bucket_num := hash (entry -> .name)
    entry -> .next := symtable [hash]
    symtable [bucket_num] := entry
    entry := next_entry
  end loop

  entry := Self.deleted
  while entry <> NIL loop
    next_entry := entry -> .next
    bucket_num := hash (entry -> .name)
    place      := symtable [bucket_num]
    loop      ! find the entry in the permanent symbol table and remove it
      if place = NIL then ! not there, so don't worry about it
        exit .
      elsif place -> .name = entry -> .name then ! got it
        if place = symtable [bucket_num] then ! at start of bucket
          symtable [bucket_num] := place -> .next
        else
          prev_place -> .next := place -> .next
        end if
        exit .
      else
        prev_place := place
        place      := place -> .next
      end if
    end loop
    entry := next_entry
  end loop
end procedure ! sym_top_precommit !

.....

merge_lock : lock ( busy : [] )

procedure merge_state (! partner_version : version_vector !) is action
! Same as in Example 1.

procedure reinit () is action
! Same as in Example 1.

procedure set_partner (! p : symtab, rep_number : replicate_range !) is
! Same as in Example 1.

```

```
begin ! initialization section
  here      := replicate_number
  master    := here = 1 ! arbitrary choice
  local_version := version_vector"[ 0 : MAXREPLICATE ]
  symtab    := symtab_type"[ NIL : MAXBUCKET ]
              ! symbol table is initially empty
end implementation.
```

### 3.2.3. The Permanent Heap

The design of the SYMTAB object presented in the preceding subsection requires the use of linked lists allocated in a heap in the permanent area of per-object storage, both for its per-action and permanent variables. This *permanent heap* will require special run-time support for its management, which must maintain the heap's consistency across failures.

In Example 3, we show a preliminary design for the permanent heap manager. To maintain the consistency of the heap, this PERMHEAP object uses the same techniques which we used in the SYMTAB object of Example 2 to implement recovery for the symbol table data structure, i.e., per-action variables and associated action-event handlers. (In fact, due to its compactness, this example may demonstrate the use of these techniques more clearly than does the SYMTAB example.) The actual management of memory is done by a HEAP object (whose definition is shown in Example 3 for clarity), which can allocate and free blocks of memory in both the permanent and the temporary heap areas. The HEAP object does not implement recoverability at present; however, once the PERMHEAP object is available, the HEAP object may be altered to use the permanent heap for its FREE list and bootstrapped.

The PERMHEAP object maintains lists of those areas of the heap allocated and freed by each action, in per-action variables. Since the HEAP object (which is at present nonrecoverable) does the actual management of the heap, allocations are visible to other actions immediately, thus maintaining the consistency of the heap. A call to the ALLOCATE operation of PERMHEAP will return a pointer to a block of memory allocated by HEAP in the permanent heap area of the object; a pointer to the block is also added to the ALLOCATED per-action list. A call to PERMHEAP's FREE operation will actually dispose the block of memory only if it was allocated by the action which is trying to free it; otherwise, a pointer to the block to be disposed is merely added to the FREED per-action list. Upon abort of an action which allocated permanent heap storage, the ALLOCATED list is used to clean up the heap via calls to HEAP's FREE operation. When a nested action enters its commit phase, its ALLOCATED and FREED per-action lists are propagated to its parent. Memory blocks on the permanent heap allocated by an action are actually disposed when the action's toplevel ancestor (to which the nested action's per-action lists have been propagated) enters its precommit phase; this is done by invoking the FREE operation of HEAP on all members of the toplevel action's FREED list:

Note that this implementation of the PERMHEAP object does not provide strict serializability. To see this, consider some action, A, which exhausts (or nearly exhausts) the permanent heap, causing other actions B and C trying to allocate permanent memory to fail. Action A may well be aborted itself. Actions B and C which failed because of A might not have failed had they been executed serially. However, such breaches of strict serializability do not affect the consistency of the permanent heap mechanism, and thus are of little concern in this context.

*Example 3. Run-time support for the permanent heap*

```
implementation of object permheap is
! Support for the permanent heap, using per-action variables for
! recovery management.

uses heap
! The definition of the HEAP pseudo-object is shown here for clarity.
! The HEAP object implements a standard heap management discipline (i.e.,
! without recovery), but allows one to allocate memory in either the
! permanent or the temporary memory area.
!
! definition of object heap is
!   type heap_type is ( normal_heap, permanent_heap )
!   operations
!     procedure allocate ( size : unsigned ,
!                         kind : heap_type ) : address
!       -- the ALLOCATE operation returns a pointer to a block of
!       -- memory of the specified SIZE in the area of memory
!       -- indicated by KIND.
!     procedure free ( block : address )
!       -- the FREE operation disposes the block of memory pointed
!       -- to by BLOCK.
!   end definition.

! The local declarations of the PERMHEAP object.
!
! Give the names of alternate handlers for some of the action events.

action events
  abort is permheap_abort,
  nested_commit is permheap_nested_commit,
  toplevel_precommit is permheap_top_precommit

! The BLOCKLIST type is used in the declaration of the per-action variables
! below.

type ptr_blocklist is -> blocklist

type blocklist is
  record
    block : address,
    next : ptr_blocklist
  end record

! The per-action variables for permanent-heap recovery management.
! We will maintain lists of memory blocks allocated and freed by each action.

per_action is
  record
    allocated, freed : ptr_blocklist
  end record
  init peraction"[ NIL:2 ]
```

```

!
! The operations of the PERMHEAP object.
!

```

```

procedure allocate (! size : unsigned ! ) ! : address ! is
! Return a pointer to a block of memory of the given SIZE in
! permanent memory.

list : ptr_blocklist

begin
  new (list)          ! create a new entry for the ALLOCATED list
  using l := list -> do
    l.block          := heap @ allocate (size, permanent_heap)
    l.next           := self.allocated
    Self.allocated := list ! put new entry at beginning of ALLOCATED list
    return l.block
  end using
end procedure ! allocate !

```

```

procedure free (! block : address ! ) is
! Dispose the block of memory indicated by BLDCK.

prev, list : ptr_blocklist

begin
  list, prev := Self.allocated ! First, scan the ALLOCATED list to see if
  loop          ! BLDCK was allocated by the current action
    if list = NIL then          ! Nope, go below
      exit .
    elsif list -> .block = block then ! Yes, so
      if prev = next then      ! remove it from ALLOCATED list;
        Self.allocated := NIL
      else
        prev -> .next := list -> .next
      end if
      heap @ free (list -> .block) ! go ahead and dispose it
      dispose (list)
      return .                    ! we're done
    else
      prev := list
      list := list -> .next
    end if
  end loop
  new (list)          ! If we get here, BLOCK wasn't allocated by the
  using l := list -> do ! current action, so put it on the FREED list
    l.block := block
    l.next := Self.freed
  end using
  Self.freed := list
end procedure ! free !

```

```

procedure permheap_abort () is
! The alternate handler for the ABORT action event.
! We'll just free all the space allocated by this action as indicated
! by the ALLOCATED list, and clean up the FREED list for good measure.

list, old : ptr_blocklist

begin
list := Self.allocated
while list <> NIL loop
heap @ free (list -> .block)
old := list
list := list -> .next
dispose (old)
end loop

list := Self.freed
while list <> NIL loop
old := list
list := list -> .next
dispose (old)
end loop
end procedure ! permheap_abort !

procedure permheap_nested_commit () is
! The alternate handler for the NESTED_COMMIT action event.
! We'll propagate the items on the ALLOCATED and FREED lists of this
! action to the beginning of the corresponding lists of its parent action.

list : ptr_blocklist

begin
list := Self.allocated
if list <> NIL then
loop ! find the end of the ALLOCATED list
if list -> .next = NIL then
exit .
end if
list := list -> .next
end loop
list -> .next := Parent.allocated
Parent.allocated := list
end if

list := Self.freed
if list <> NIL then
loop ! find the end of the DELETED list
if list -> .next = NIL then
exit .
end if
list := list -> .next
end loop
list -> .next := Parent.freed
Parent.freed := list
end if
end procedure ! permheap_nested_commit !

```

```

procedure permheap_top_precommit () is
  ! The alternate handler for the TOPLEVEL_PRECOMMIT action event.
  ! We'll use the normal HEAP operation FREE to dispose of the memory blocks
  ! on the FREED list, but we'll just dispose the ALLOCATED list -- it's only
  ! used to free up storage allocated by an aborting action.

  list, old : ptr_blocklist

begin
  list := Self.freed
  while list <> NIL loop
    heap @ free (list -> .block)
    old := list
    list := list -> .next
    dispose (old)
  end loop

  list := Self.allocated
  while list <> NIL loop
    old := list
    list := list -> .next
    dispose (old)
  end loop
end procedure ! permheap_top_precommit !

begin ! Object initialization
  null
end implementation.

```

#### 4. RELATED WORK

As with most of the topics involved in the study of distributed systems, the synchronization and recovery of replicated data was first studied in the area of distributed database systems. The history of these efforts is summarized by Wright [Wrig83]. He classifies these methods as *conservative* (*pessimistic*, *blocking*) and *optimistic* (*non-blocking*). Examples of conservative methods are voting schemes [Giff79, Thom78], primary copy methods [Ston79], and token-passing schemes [LeLa78]. The intent of these methods is to ensure consistency of the replicated data by requiring access to a special copy or set of copies of the data during partitions. Primary copy methods allow access to a copy during a network partition only if the partition possesses the designated primary copy of the data. Token-passing schemes are an extension of primary copy methods; a token is passed among sites holding a copy of data, and that copy at the site currently holding the token is considered the primary copy. Yet another extension of primary copy methods are the voting schemes. Each copy of the data object is assigned a (possibly different) number of votes; a partition possessing a majority of the votes for that object may access it. The conservative schemes are called *blocking* since a data object 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 object.

The optimistic methods do not seek to ensure global consistency of replicated data during partitions [Davi81, Davi82]. Thus, accesses are not blocked if a replicate of the data is available in the partition in question. Rather, inconsistencies in the data replicates are resolved during a merge process once the partition is ended, by use of backouts or compensatory actions. 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.

Wright develops enhancements to both the conservative and the optimistic methods. Conservative schemes are extended by the notion of compatibility among classes of transactions, which allow increased efficiency and availability with these methods of concurrency control. He also considers the computational complexity of the problem of backing out transactions under optimistic schemes, and shows that (in general) the problem is NP-complete. He then develops efficient heuristic solutions to this problem.

However, Wright's work (as is most of the work previously discussed) assumes a simple data model based on reads and writes. He does speculate that the semantic knowledge about objects available in object-oriented systems may bring about the possibility of gains in efficiency over his model, since the read-write model places unnecessary restrictions on availability and concurrency when used with more complex objects. The *Data-Patch* tool mentioned above takes advantage of semantic knowledge through its YACC-like approach to the construction of partition-merge routines for databases.

Previous work in the area of replication of data in distributed operating systems includes work on the LOCUS system at UCLA [Walk83] as well as the Argus system at MIT [Herl84] and the ISIS system at Cornell [Birm84]. The LOCUS system supports replicated files and directories using an optimistic approach; inconsistencies are allowed to develop among the separate partitions which are resolved (except in the case of simple read-write file objects) by application-dependent measures. No mention is made of system support for the applications' recovery methods. Herlihy's work at MIT uses semantic knowledge of Argus objects to enhance a conservative (voting) method. Analysis of the algebraic structure of data types is used in the choice of appropriate intersections of voting quorums. The ISIS system supports *k-resilient* objects (objects replicated at  $k+1$  sites and which can withstand up to  $k$  failures) by means of checkpoints and the "available copies" voting algorithm. This system provides both availability and *forward progress*, that is, even after up to  $k$  site failures, enough information is available at the remaining sites possessing an object replicate that work started at the failed sites can continue at these remaining sites. This is accomplished through a *coordinator-cohort* scheme similar to the master-slave discipline shown in the previous section.

Thus, recalling the issues detailed in Section 3.1, we believe that the proposed research will lead to contributions in several of the areas mentioned. The use of semantic knowledge of objects in the programming of non-blocking replication methods has not been the subject of much previous study, especially in the context of systems programming problems. The trade-offs mentioned, between consistency and availability and between blocking and non-blocking replication methods, have been the focus of some work in the database realm, but again the issues have not been treated in operating systems. The issue of appropriate programming models for availability in action/object systems has not been treated before. Rather, in systems such as Argus, the applications language has been designed *ab initio*, and the system as well as programming models for it have been cut to fit. Finally, the study of support needed for availability and for forward progress should provide valuable insights.

## 5. OUTLINE OF DISSERTATION

A proposed outline for the dissertation resulting from the proposed research is presented here. Since this is an exploratory thesis, the answer to the perennial question "how can we tell when you are done?" is a difficult one. There will be some tangibles; in particular, designs for the run-time support system for Aeolus and for the interfaces to the action manager, object manager, and job scheduling systems should be forthcoming. However, completion of several portions of the work will depend on our satisfaction with the completeness of the set of case studies and with the insights into the design of language features which these case studies may yield.

### *Introduction*

Background and terminology for the research to be discussed will be presented in terms of an overview of the Clouds project and of the Aeolus language. The goals and plan of the research will be described.

### *Contributions*

The contributions of the research will be summarized.

### *Related Work*

Previous work in this area will be discussed and compared to this research.

### *Case Studies*

The results of the explorations in programming methodology for replicated data will be presented and discussed.

### *Language Features for Resilience and Availability*

Those features whose designs result from the case studies will be presented and discussed; in particular, comparisons will be made with features provided in other languages for distributed applications.

### *Run-Time and Operating System Support for Replicated Data*

Designs or suggestions for support features resulting from the case studies will be presented and compared to support provided by other systems.

### *Conclusions and Further Work*

The work done and its contributions are summarized; ideas for further work beyond the scope of this research which may develop are presented.

## 6. REFERENCES

- [Allc82] Allchin, J. E., and M. S. McKendry, "Object-Based Synchronization and Recovery," Technical Report GIT-ICS-82/15, School of Information and Computer Science, Georgia Institute of Technology, September 1982
- [Allc83a] Allchin, J. E., and M. S. McKendry, "Synchronization and Recovery of Actions," *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, August 1983
- [Allc83b] Allchin, J. E., "An Architecture for Reliable Decentralized Systems," Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, 1983 (also available as technical report GIT-ICS-83/23)
- [Birm84] Birman, K. P., T. A. Joseph, T. Raeuchle, and A. El Abbadl, "Implementing Fault-Tolerant Distributed Objects," *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, Silver Spring, Maryland, October 1984
- [Blau82] Blaustein, B., R. M. Chilenskas, H. Garcia-Molina, D. R. Ries, and T. Allen, "Partition Recovery Using Semantic Knowledge," Computer Corporation of America, Cambridge, Massachusetts, November 1982
- [Dani83] Daniels, D., and A. Z. Spector, "An Algorithm for Replicated Directories," *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, August 1983
- [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, Pennsylvania, July 1981
- [Davi82] Davidson, S., "An Optimistic Protocol for Partitioned Distributed Database Systems," Ph.D. Thesis, 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, Florida, October 1983
- [Giff79] Gifford, D. K., "Weighted Voting for Replicated Data," *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, Pacific Grove, California, December 1979
- [Herl84] Herlihy, M. P., "Replication Methods for Abstract Data Types," Ph.D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, May 1984 (available as MIT/LCS/TR-319)
- [LeB185] LeBlanc, R. J., and C. T. Wilkes, "Systems Programming with Objects and Actions," to appear in *Proceedings of the Fifth International Conference on Distributed Computing Systems*, Denver, Colorado, May 1985 (also available as Technical Report GIT-ICS-85/03)
- [LeLa78] LeLann, G., "Algorithms for Distributed Data-Sharing Systems which use Tickets," *Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks*, Berkeley, California, August 1978
- [McKe84a] McKendry, M. S., "Clouds: A Fault-Tolerant Distributed Operating System," School of Information and Computer Science, Georgia Institute of Technology, May 1984
- [McKe84b] McKendry, M. S., "Ordering Actions for Visibility," *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, Silver Spring, Maryland, October 1984 (also available as Technical Report GIT-ICS-84/05)

- [Pitt84] Pitts, D., "Storage Management for an Action-Based Operating System," Ph.D. Thesis Proposal, School of Information and Computer Science, Georgia Institute of Technology, November 1984 (also available as Technical Report GIT-ICS-85/02)
- [Ston79] Stonebreaker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Transactions on Software Engineering* 5, 3, May 1979
- [Thom79] Thomas, R. H., "A Majority Consensus Approach to Concurrency Control for Multiple-Copy Databases," *ACM Transactions on Database Systems* 4, 2, June 1979
- [Walk83] Walker, B., G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, Bretton Woods, New Hampshire, October 1983 (available as *Operating Systems Review* 17, 5)
- [Wrig83] Wright, D. D., "Managing Distributed Databases in Partitioned Networks," Ph.D. Thesis, Department of Computer Science, Cornell University, January 1984 (available as Technical Report 83-572, September 1983)