Transparent Distribution Using Two Object Granularities

Frans A. Henskens[o], Peter Brössler[†], J. Leslie Keedy[†] & John Rosenberg[o]

[†] Fachbereich 3 Informatik
Universität Bremen
Postfach 330440
2800 Bremen
Germany
Ph: +49 421 218 2920
Fax: +49 421 218 4269
Email: {pb,keedy}@informatik.uni-Bremen.de

[o] Basser Department of Computer Science
University of Sydney
N.S.W. 2006
Australia

Ph: +61 2 692 3423
Fax: +61 2 692 3838
Email: {frans,johnr}@cs.su.oz.au

Abstract

This paper describes our experiences with the development of a Distributed Shared Memory (DSM) based on a single, very large, paged virtual memory space distributed across an arbitrary number of discrete nodes connected to a network of homogeneous computers. The DSM supports two object granularities: coarse-grain objects called modules and fine-grain objects called segments. We show that support for both modules and segments has advantages in the areas of naming and location, protection, data consistency, transaction management and garbage collection.

1 Introduction

This paper describes our experiences with the development of a Distributed Shared Memory (DSM) based on a single, very large, paged virtual memory space distributed across an arbitrary number of discrete nodes connected to a network of homogeneous computers. DSMs allow processes executing on loosely coupled nodes to share data by reference, such that knowledge of the address of data in the shared memory space is sufficient to allow a process to access the data. The DSM paradigm has been implemented by other researchers to form shared memory systems, for example Ivy [20] and Memnet [8]. The main aim of these implementations was the exploitation of parallel algorithms on loosely-coupled processors, but their architectures were not expandable to large shared virtual memories, and were not oriented towards supporting persistence.

The DSM described here, on the other hand, implements a global distributed store, enabling users logged on to loosely-coupled computers to share all resources including programs, data, and devices. The DSM has several important properties relating to the user's perception of the services provided by a networked node compared to those offered by a discrete node. These are:

(1) The distribution of the system provides extra functionality without compromising the functionality provided by a discrete, non-networked machine.

(2) A resource is identified by name. This name defines the resource only, and not its current location. As a consequence the resource may be moved to another node and still be accessed using its original name.

(3) All users at all nodes have a coherent view of shared data.

(4) The owner of a resource has control over access to the resource on a network-wide basis.

(5) Programs developed for use on a non-networked machine execute without modification over the network.

Published in: *Proceedings, European Conference on Object-Oriented Programming*, Kaiserslauten, Germany, pp. 37-47, 1993.

Such an implementation of DSM provides a single network-wide virtual memory space in which all program code and data is stored. This virtual memory provides a store which complies with the orthogonal persistence principles of persistence independence, data type independence, and management orthogonality [2]. Since all processes execute within this virtual memory space, the conventional process address space protection scheme used by systems such as Unix cannot be used. A two-level capability-based protection technique enforced by the architecture is used to provide control over access to the programs and data in the store [17].

The system supports two object granularities: coarse-grain objects called modules (which are roughly equivalent to files in conventional systems) and fine-grain objects called segments (containing for instance, procedures, records, characters or integers) [6]. Programmers and compilers see the virtual memory as a collection of segments of arbitrary size. These segments are mapped onto the paged virtual address space in such a way that page and segment boundaries are orthogonal [16]. All segments have the same basic format, so access to them is handled in a uniform manner. Segments contain data, and capabilities for other segments, allowing arbitrarily complex graph structures of segments to be constructed. Access to a segment is controlled by a segment capability which defines the segment start address, length, and type and access information.

Segments are grouped together into information-hiding modules which present a purely procedural interface**,** as proposed by Parnas [23]. Access to a module is controlled by a module capability which contains a unique network-wide name for the module and a set of access rights for the interface procedures which may be invoked when the capability is presented. Such procedures in turn may access the encapsulated data segments.

Processes are orthogonal to modules. A process may call an arbitrary number of modules and such calls may be arbitrarily nested. Similarly, many processes may concurrently execute within any one module. This structure is fully described in [18].

The DSM virtual memory space is of sufficient size to obviate the need ever to re-use the ranges of addresses previously allocated to deleted modules. Hence the address of a module may be used as the module's name, uniquely identifying it and the code or data segments encapsulated by it.

Support of two object granularities has advantages in the areas of:

- naming and location
- protection
- data consistency
- transaction management
- garbage collection

In this paper we first describe the general DSM architecture and the relevant implementation details. We then address in more detail the five areas listed above and discuss the implication of dual object sizes on issues including performance, flexibility and ease of implementation.

2 Overview of DSM Architecture

The DSM [15], which provides a distributed persistent store, was constructed above a purpose-built microcoded computer, known as Monads-PC, which provides a 60 bit wide paged virtual memory [24]. Both diskless nodes and nodes with attached disks are supported. Nodes with disks act as servers for the pages of the objects stored on those disks.

In this section we describe the structure of the virtual memory and how this may be accessed in a distributed environment.

2.1 Memory Management

Two mappings are required in order to implement virtual memory. The first is a mapping from virtual addresses to main memory addresses for pages currently in main memory, and the second is a mapping from virtual addresses to disk addresses for pages not in main memory. The conventional approach to virtual memory uses the same data structures and mechanisms, based on page tables, for both of these mappings. Our model, on the other hand, decouples the virtual address to main memory address mappings (which are needed for every memory reference) from the virtual address to disk address mappings (which are only needed in page fault resolution) [26] The importance of this for distribution is that each node maintains a main memory page table proportional in size to the size of its own main memory, and the disk page tables for disks mounted at the node. In contrast other DSM implementations (eg [8, 20]) require a data structure defining the status and location of every virtual memory page to be maintained by every node. Such structures are proportional in size to that of the distributed virtual memory.

To minimise the size of these page tables, other DSM systems partition the address space in which a process executes into local and shared areas. Only the shared memory partition is implemented as DSM. Using this technique, the developers of these systems were able to simplify communication between parallel processes, but could not provide totally transparent distribution of all data. The method used to manage the virtual memory in our architecture allows the entire virtual memory space to be implemented as DSM, distributed across a network of nodes.

In the Monads-DSM system the virtual memory encompasses the attached disks and main memory of all machines network-wide. Thus virtual addresses can refer to any byte on any disk connected to any node. In order to resolve a page fault, the disk location of the page must be determined. The first step involves determining which of the attached disks contains the page.

Every node is assigned a unique node number when it is manufactured. This is a logical node identifier, and is not the physical network address of the node. Disk drives attached to the nodes may be partitioned as part of the formatting operation, thus creating several logical disks on a single physical device. Each of these partitions is known as a volume. When a volume is created it is assigned a unique node+volume number which is formed by concatenating the creating node number with the within-node volume number. This node+volume number is used to define the range of virtual addresses that is stored on the volume by using it as the high order bits of all such addresses.

The range of addresses stored on a volume is further divided into areas corresponding to the logical entities such as processes, files and programs that exist on the disk. These areas are called address spaces, and are identified by address space numbers. Address spaces are further divided into fixed size pages identified by page numbers. A virtual address, then, consists of five parts, as shown in figure 1.

Figure 1. The Structure of a DSM Virtual Address.

All the pages of an address space are stored on a single volume. Each address space has its own disk page table which maps from virtual addresses to disk addresses for that address space. This table is contained within the address space and pointed to from the root page of the address space. Thus every address space is self-defining, and efficient use is made of disk space because disk

pages are not allocated to unused virtual pages. Address space zero for each volume is special. It contains red-tape information for the volume, including the free space map and the volume address space table listing the disk locations of the root page for each address space on the volume. Each module and stack (which represents a process) resides in a separate and unique address space, and is named according to the identity of the address space within which it resides. An address space is never re-used, even if the module or stack residing in it is deleted, so the name of a module or stack is unique for the life of the system. Access to a module is permitted on presentation of a valid module capability. The architecture protects module capabilities from illegal modification or use. The address space number for a module is embedded in the module capability used to access it, together with other fields which define the nature of the access permitted.

Programmers and compilers see the virtual memory as a collection of segments which may be of arbitrary size, from one byte to 256 Mbytes in the Monads PC. Because segment boundaries are orthogonal to page boundaries [16], excessive internal fragmentation is avoided. Segments contain data and capabilities for other segments, with the result that complex graph structures can be constructed. Access to a segment is permitted on presentation of a valid segment capability. This capability defines the full virtual address of the start of the segment and the segment length and is protected from arbitrary modification. The basic addressing mode supported by the architecture involves the specification of an offset relative to a segment capability.

Segments are grouped together into the previously described modules. Presentation of a valid module capability allows a process to open the module, after which the segment capabilities which allow access to its interface routines and internal data become available. These segment capabilities are stored in a special module call segment (MCS) which is initialised after access to the module root page. The significance of these structures will become apparent in the following sections.

2.2 Distributed Access

The DSM model is based on a single very large virtual memory space which encompasses all nodes in the network. Processes running on these nodes have access to the whole virtual memory space (provided they can present an appropriate capability), without the need for knowledge of the storage location of the program code and data they access. This model was initially proposed in [1], Related schemes have been reported in the literature [8, 20]. However these schemes allow processes to share only a limited portion of their total address space, and still maintain a separate file store.

The Monads DSM is designed to support the interconnection of Monads-PC computers using a local area network (LAN) [13]. The kernel at each machine maintains knowledge of the mappings between the network addresses of connected nodes and their Monads node numbers using an up/down protocol similar to ARP/RARP [7]. Since the processors are loosely coupled, there is no physically shared memory, so the underlying communications system is used to provide an abstraction of a shared memory space.

During execution a process accesses a sequence of virtual addresses. If the virtual page containing such an address is in the local node's page cache (main memory), the access may proceed. If not, a page fault condition applies. To resolve the page fault, the local kernel examines the <node number><volume number><address space> fields of the faulting address, and by consulting internally maintained tables it determines whether the page fault may be resolved by a local disk access. If not, the kernel causes transmission of a message requesting provision of the page. In this sense each node views the other nodes in the network as collections of volumes.

3 Naming and Location Transparency

As described in section 2.1, a module is named according to the address space in which it resides. This defines the position of the module in the virtual memory space. The module name is embedded in the capabilities used to address the module. Any node with attached disks acts as a server for the pages of the modules stored on those disks. During the life of the system, it may become necessary to mount disks on different nodes. For example a node may fail; mounting its disks on another node would make the data on those disks available. It would also be beneficial to efficient use of network bandwidth to move the modules owned by a user to his new home node when his place of work changes. As described in section 2.2, the module location information embedded in addresses is crucial to efficient access to their pages. Module addresses form part of the capabilities used to control access to them. Since no attempt is made to record the owners of such capabilities, it is important that the name of a module is not modified when the storage location for the module changes. If this were not done, the movement of a module would render invalid all existing capabilities for it. A change of storage location for a module occurs if either

(1) the volume containing the module is mounted on another node, or

(2) the module is moved to another volume.

3.1 Moving Volumes

Page request messages are typically transmitted to the node whose identity is embedded in the page address. Prior to transmitting a request for provision of a page, however, the kernel checks local tables which map moved volumes to their new mounting node. These tables are maintained on a need to know basis using a kernel message protocol. If necessary the destination node for the page request message is adjusted accordingly.

3.2 Locating Moved Modules

Access to a module which has been moved between volumes is detected when the module is opened. Advisory information regarding the new location is obtained from the module capability and used to direct the page request message. If such advisory information proves to be incorrect, the creating node is queried, and if possible it provides forwarding information maintained on the original storage volume. In the case that neither of these attempts is successful, a broadcast message is used in an attempt to locate the module. Location of a moved module therefore incurs an overhead compared to the location of a non-moved module.

Once the module has been located and opened, subsequent page requests occur as a result of page faults generated by accesses to the segments within the module. Since such page requests occur much more frequently than open module requests it is important that location of the server node is efficient.

At the time of such page faults all that is available to the page fault handler is the faulting virtual address, which contains the original location of the module. The kernel could maintain a table of moved modules and their current locations [3]. However, this table would have to be searched on every page fault, resulting in an overhead for all requests, including those for non-moved modules. We therefore use a different approach as described in the following section.

3.3 Efficient Access to Moved Modules

The essence of the technique presented in this section is that the identity of an open module may be temporarily altered to reflect its current location, thus allowing efficient access to the pages of the module [12].

The implementation of this technique for accessing moved modules requires that:

(1) A new unique address space number defining the new node, volume, and (logical) within-volume address space is allocated to a moved module. This number is called the current name for the module, and is used for internal system purposes only. The name by which the module is known to users remains the name allocated when the module was created, and as a result all existing module capabilities still allow access to the module. The current name may be viewed as an alias for the original name.

(2) An additional table is maintained in the red tape of each volume. This table is called the Foreign Address Space Table (FAST), and contains mappings between module names and current names for moved modules currently stored on the volume, as shown in figure 2. The FAST is accessed using the original module name as a key, and allows the current name for any moved module stored on the volume to be determined.

(3) When a module is moved its volume directory entry at the new owner node contains the current name for the module.

(4) When a module is moved from a volume the volume directory of the source volume is changed to link the current name used on the source volume to a forwarding address in the same manner as described in the previous section.

(5) The system is able to detect that a newly opened module has been moved from its original storage volume.

When a module is opened, the root page of the module must be accessed to allow creation of the MCS. When the MCS is set up the system determines whether the module is stored on its original volume or has been moved to a different volume. If the module has been moved, the system is informed of its new location when it obtains the module's root page.

Efficient access to the pages of the module is achieved by altering the segment capabilities used to access the module's data as they are stored in the MCS. This alteration replaces the original node number, volume number, and address space number fields with values indicating the current node, volume, and address space numbers. Subsequent accesses to these data segments generate virtual addresses containing the current name rather than the original name; as a result the kernel can obtain pages from these segments as if the module had never been moved. Since all the segments of a module are contained within the module, and pointers to data within a module are relative to the address space in which it resides, these pointers do not need to be changed as a result of the change of address space name.

Module Name	Current Name
Creating Node, Volume, and Address Space Number of Module	Current Node, Volume, and (Logical) Address Space Number of Module

Figure 2. The Structure of a Typical Foreign Address Space Table Entry.

Significantly, the use of a Foreign Address Space Table (FAST) in accessing the segments stored in moved modules allows page faults for accesses to such segments to be resolved efficiently

whilst not increasing the overhead of resolution of page faults for segments stored in non-moved modules.

The encapsulation of small objects (segments) within large objects (modules) has allowed us to incur all of the overheads of locating moved (and non-moved) objects at the time at which a module is opened, rather than the time at which page faults occur. Thus the use of two object sizes has provided a considerable performance advantage.

4 Protection

In a conventional distributed system protection is provided by a combination of a separate addressing environment for each process and the file system. In a distributed shared memory such as that described above the addressing environment for every process is the entire shared memory. As a result, in the absence of some additional protection mechanism, every process has access to every byte of data stored in the network. It is therefore necessary to provide a protection mechanism to control access to data. Given that data is logically grouped into structures such as arrays, procedures, etc., it would seem appropriate to support protection at this (small object) level.

A suitable means for providing such protection is the use of capabilities [9]. We have described above that access to segments is controlled by segment capabilities. The right to access a segment must be checked on every access. It is therefore essential that the mechanism be simple and efficient. In the case of our DSM this has been achieved by provision of hardware support for capability-based addressing [24].

There are clearly advantages from a software engineering point of view in support for information hiding modules. Such modules hide implementation details and therefore simplify maintenance and improve reliability. They encourage the construction software of systems in a modular fashion [23]. However, in many object-based systems this encapsulation may be circumvented and object data directly manipulated.

Because we provide direct support for coarse-grain objects it is possible to guarantee the integrity of modules by enforcing that access to their data is through the provided interface routines [19]. This is achieved by supporting a second level of capability, called a module capability. A module capability includes the name of the module referenced and a list of the interface procedures (methods) which may be accessed by the holder of the capability. This is implemented by storing each interface procedure in a separate segment and ensuring that only appropriate segment capabilities are made accessible to the process at the time the module is opened.

The provision of such flexible protection for modules without sacrificing efficient access at the data structure level is only possible because of the separation of the two granularities of object. In a system which supports only one granularity of object, all of the overheads of method invocation are incurred for all object accesses [10].

5 Consistency

A major issue in the design of a global DSM for a persistent environment is the ability to recover a consistent state after a crash. According to the DSM protocol, current versions of parts of an object may be spread over several nodes in the network. The failure of any of such nodes can result in the loss of parts of the current version of the object, with a subsequent loss of object integrity. In addition, modifications to other objects may have been based on the lost changes. Although these additional objects may be self-consistent after some failure, they may not be consistent with data contained in other objects [14].

A technique for solving this problem is to ensure that the store moves from one consistent state to another. This can be achieved by periodically copying the entire store to a stable medium, such as disk. This process is usually called checkpointing and a number of proposals for efficient

checkpointing have been described in the literature [5, 21, 25]. During such a checkpoint all operation on the store must cease.

In many circumstances checkpointing the entire store as a single operation may have unacceptable performance implications, particularly for a distributed store supporting concurrent access. A better approach is to checkpoint regions of the store independently [14]. However, as we have indicated above, there may be relationships between objects within the store. It is essential that related objects are checkpointed at the same time to ensure that they are consistent with each other. Such groups of related objects have been referred to as associations [28].

Maintaining associations based on fine-grain objects rapidly results in large associated sets. Given that we have architectural support for grouping related fine-grain objects to form coarse-grain objects, it becomes possible to maintain associations at the coarse-grain level. These associated sets are considerably smaller than those for fine-grain objects. Performance is improved for two reasons. First, the data structures used to store association information are significantly reduced in size. Second, coarse-grain objects may be checkpointed at the virtual page level, with each write operation on a virtual page potentially checkpointing multiple fine-grain objects.

6 Transactions

The two different object granularities are also beneficial for transaction processing [11]. The basic form of (nested) transactions uses segments. Applications requiring a higher degree of parallelism than is possible without semantic knowledge use object-level transactions.

Basic transactions [4] use segments as the entities for concurrency control and recovery. When a transaction loads a segment capability for the first time, the segment is read-locked. The lock data structures are part of the segments thus simplifying the management and the addressing of the lock information. The first update of a segment within a transaction leads to the acquisition of a write-lock and to the writing of a before-image into a specific address space. A commit of a transaction releases all the locks and before-images, whereas an abort restores all segments to their original values and then releases all locks.

Object-transactions allow a higher degree of parallelism by using commutativity relationships between routines of objects. Each call to such a routine is executed as a basic transaction. Segment-level locks are released at the end of such a call and routine-specific (semantic) locks are held until the end of the whole transaction. The execution model follows the concept of generalised multi-level transactions [22]. Aborts of object-transactions must be handled by the execution of logical undo operations in contrast to the restoration of segments in basic transactions.

7 Garbage Collection

Garbage collection is an important issue for single level stores such as DSMs. In most systems objects contain both data and references to other objects. Garbage collection involves traversing the graph of objects from some root and removing all unreachable objects. This is of particular importance in DSM systems because it releases parts of the store for re-use. The disk space associated with garbage is also released for re-use.

There are two basic approaches to garbage collection:

1. Garbage collection of the entire store as a single operation. This is extremely expensive for large distributed stores and may not be possible, if for example, one of the nodes is unavailable.
2. Garbage collect regions of the store independently. This reduces the impact of garbage collection on users of the store.

The second approach seems more appealing. The issue is the criterion for the division of the store into regions. A common approach is to divide into regions based on time of object creation. Such

collectors are usually called generation-based garbage collectors. Proponents of this technique argue that most recently created objects are temporary and will soon become garbage. They segregate such objects and maintain a table of references between the segregated region and older objects. However, eventually the entire store must be garbage collected in order to remove unreachable older objects [27].

An alternative is to group objects according to their logical relationships. Our two level coarse and fine grain model effectively provides such a grouping automatically. No additional data needs to be maintained in order to garbage collect such groups. In our DSM, references between fine-grain objects within different coarse-grain objects are prohibited. Therefore it is possible to garbage collect each coarse-grain object independently. If required it is possible to perform generation-based garbage collection within modules.

8 Conclusion

Figure 3 contains a summary of the most significant differences between coarse-grain and fine-grain objects in the Monads DSM.

Coarse-Grain Objects: Modules	Fine-Grain Objects: Segments
major system objects	components of major objects
enforcement of information hiding	programming language-specific usage
user-level protection: module capabilities	system-level protection: segment capabilities
virtual address space allocation	heap space allocation
size between 1 and 65536 pages	size between 1 byte and 256M-bytes
independent placement and migration	clustered within large object
long unique object ids	short reusable pointers
cross references to other large objects	local references to segments within same large object
explicit deletion rule based on ownership	automatic garbage collection based on reachability
flexible semantic concurrency control	efficient transactional read/write concurrency control
transaction recovery based on inverse operations	transaction recovery based on before images

Figure 3. A Summary of Differences between Coarse and Fine Grain Objects.

We have described a system based on a distributed shared memory which supports two granularities of object. Segments are fine-grain objects which are used to hold logical entities such as structures, arrays, procedures, etc. Logically related segments are grouped together to form coarse-grain information-hiding objects called modules.

It was shown that there are significant advantages in supporting two granularities of object. First, coarse-grain objects provide a convenient and efficient mechanism for naming and locating data in a distributed system. Second, different protection paradigms may be supported for each object

granularity. This allows efficient access for fine-grain objects without sacrificing flexibility for coarse-grain objects. Third, coarse-grain objects assist with limiting the cost of checkpointing by allowing groups of objects to be checkpointed independently. Fourth, fine-grain objects are used for efficient transaction management whereas course-grain objects allow increased parallelism using object transactions. Finally, coarse-grain objects provide an appropriate clustering of fine-grain objects for the purposes of garbage collection.

Most of the mechanisms described in this paper have been implemented in a distributed network of Monads-PC computers. Work is continuing in the areas of distributed checkpointing mechanisms and object availability techniques.

Acknowledgments

Part of this work was supported by a University of Sydney research grant.

References

1. Abramson, D. A. and Keedy, J. L. "Implementing a Large Virtual Memory in a Distributed Computing System", *Proc. 18th Hawaii Conference on System Sciences*, pp. 515-522, 1985.

2. Atkinson, M. P., Bailey, P., Chisholm, K. J., Cockshott, W. P. and Morrison, R. "An Approach to Persistent Programming", *The Computer Journal*, 26, 4, Nov., pp. 360-365, 1983.

3. Brössler, P., Henskens, F. A., Keedy, J. L. and Rosenberg, J. "Addressing Objects in a Very Large Distributed System", *Proc. IFIP Conference on Distributed Systems*, North-Holland, pp. 105-116, 1987.

4. Brössler, P. and Rosenberg, J. "Support for Transactions in a Segmented Single Level Store Architecture", *Proceedings of the International Workshop on Computer Architectures to support Security and Persistence of Information*, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and British Computer Society, pp. 319-338, 1990.

5. Brown, A. L., Connor, R. C. H., Carrick, R., Dearle, A. and Morrison, R. "The Persistent Abstract Machine", Universities of Glasgow and St. Andrews, Persistent Programming Research Report PPRR-59-88, 1988.

6. Chin, R. S. and Chanson, S. T. "Distributed Object-Based Programming Systems", *ACM Computing Surveys*, 23(1), pp. 92-124, 1991.

7. Comer, D. "Internetworking With TCP/IP - Principles, Protocols and Architecture", Prentice Hall, pp. 49-63, 1988.

8. Delp, G. S. "The Architecture and Implementation of Memnet: a High-Speed Shared-Memory Computer Communication Network", University of Delaware, Udel-EE Technical Report Number 88-05-1, 1988.

9. Fabry, R. S. "Capability-Based Addressing", *Communications of the A.C.M.*, 17(7), pp. 403-412, 1974.

10. Goldberg, A. and Robson, D. "Smalltalk-80", Addison-Wesley, 1985.

11. Gray, J. and Reuter, A. "Transaction Processing: Concepts and Techniques", *Morgan Kaufmann Series in Data Management Systems*, Morgan Kaufmann, 1992.

12. Henskens, F. A. "Addressing Moved Modules in a Capability-based Distributed Shared Memory", *Proceedings of the 25th Hawaii International Conference on System Sciences*, vol 1, Hawaii, U. S. A., ed V. Milutinovic and B. D. Shriver, IEEE Computer Society Press, pp. 769-778, 1992.

13. Henskens, F. A. "A Capability-based Persistent Distributed Shared Memory", Basser Department of Computer Science, University of Sydney, Australia, Technical Report 462, ISBN 0 86758 668 0, 1993.

14. Henskens, F. A., Rosenberg, J. and Hannaford, M. R. "Stability in a Network of MONADS-PC Computers", *Proceedings of the International Workshop on Computer Architectures to support Security and Persistence of Information*, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and British Computer Society, pp. 246-256, 1990.

15. Henskens, F. A., Rosenberg, J. and Keedy, J. L. "A Capability-based Distributed Shared Memory", *Australian Computer Science Communications*, 13(1), pp. 29.1-29.12, 1991.

16. Keedy, J. L. "Paging and Small Segments: A Memory Management Model", *Proc. IFIP-80, 8th World Computer Congress*, pp. 337-342, 1980.

17. Keedy, J. L. "An Implementation of Capabilities without a Central Mapping Table", *Proc. 17th Hawaii International Conference on System Sciences*, pp. 180-185, 1984.

18. Keedy, J. L. and Rosenberg, J. "Support for Objects in the MONADS Architecture", *Proceedings of the International Workshop on Persistent Object Systems*, ed J. Rosenberg and D. M. Koch, Springer-Verlag, 1989.

19. Keedy, J. L. and Vosseberg, K. "Persistent Protected Modules and Persistent Processes as the Basis for a More Secure Operating System", *Proceedings of the 25th Hawaii International Conference on Systems Sciences*, vol 1, IEEE, Hawaii, USA, pp. 747-756, 1992.

20. Li, K. "Shared Virtual Memory on Loosely Coupled Multiprocessors", Ph.D. Thesis, Yale University, 1986.

21. Lorie, R. A. "Physical Integrity in a Large Segmented Database", *ACM Transactions on Database Systems*, 2,1, pp. 91-104, 1977.

22. Muth, P., Rakow, T. C., Weikum, G., Brössler, P. and Hasse, C. "Semantic Concurrency Control in Object-Oriented Database Systems", *Proceedings, IEEE International Conference on Data Engineering*, Wien, IEEE, 1993.

23. Parnas, D. L. "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, 15(12), pp. 1053-1058, 1972.

24. Rosenberg, J. and Abramson, D. A. "MONADS-PC: A Capability Based Workstation to Support Software Engineering", *Proc, 18th Hawaii International Conference on System Sciences*, pp. 515-522, 1985.

25. Rosenberg, J., Henskens, F. A., Brown, A. L., Morrison, R. and Munro, D. "Stability in a Persistent Store Based on a Large Virtual Memory", *Proceedings of the International Workshop on Architectural Support for Security and Persistence of Information*, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and British Computer Society, pp. 229-245, 1990.

26. Rosenberg, J., Keedy, J. L. and Abramson, D. "Addressing Large Virtual Memories", *The Computer Journal*, (to appear), 1992.

27. Ungar, D. "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm", *ACM SIGPLAN Notices*, 9(5), pp. 157-167, 1984.

28. Vaughan, F., Schunke, T., Koch, B., Dearle, A., Marlin, C. and Barter, C. "A Persistent Distributed Architecture Supported by the Mach Operating System", *Proceedings of the 1st USENIX Conference on the Mach Operating System*, pp. 123-140, 1990.