A Capability-based Distributed Shared Memory

Frans A. Henskens and John Rosenberg
Department of Electrical Engineering and Computer Science
University of Newcastle
N.S.W. 2308
Australia

J. Leslie Keedy
Faculty of Mathematics and Computer Science
University of Bremen
Postbox 330440
2800 Bremen 33
West Germany

Abstract

In this paper we describe the implementation of a local area network of experimental capability-based computers. The architecture of these computers is unusual in supporting a persistent virtual memory with a very large address size, eliminating the need for a separate file store. Because the network extends this architecture by providing a global virtual memory with addressing unique across the network, the use of the network is fully transparent and existing software does not need to be modified in order to make use of remote facilities.

1. INTRODUCTION

The MONADS Project at the University of Newcastle, Australia and at the University of Bremen, West Germany, is primarily concerned with the development of a highly secure computer system which supports a rational, engineering-like approach to the development of computer software. As a step towards this goal an unconventional and novel computer architecture [12, 13, 14, 24] has been designed and a prototype implementation of this architecture, known as MONADS-PC [22], has been completed.

The most interesting features of this architecture include:

- hardware support for a very large persistent single-level store based on a paged virtual address space,
- capability-based addressing utilising unique (forever) virtual addresses, providing fine-grain control over access to data and code,
- a segmented addressing model which efficiently supports both large and small segments and provides access to persistent and temporary objects in a uniform manner, and
- direct support for the construction of software systems as information-hiding modules.

Published in: Australian Computer Science Communications, 13(1), pp29.1-29.12, 1991

In this paper we describe an extension to the MONADS architecture which supports a local area network of MONADS-based computers. Given the uniform addressing model supported by the MONADS architecture [12, 14, 24], the natural approach is to extend the addressing scheme to encompass the entire network [2]. This has several advantages over the conventional approach to network implementation.

These are:

- the interconnection of the processing units is totally transparent to users,
- objects are persistent on a network-wide basis,
- location transparency for objects is provided, so that the name of an object does not define the node on which it resides,
- naming transparency for objects is provided, so that the same name used on different nodes will identify the same object,
- coherency of shared objects is maintained by the system, and
- the owner of an object has capability-based control over access to the object on a network wide basis.

The paper begins by describing the relevant features of the MONADS architecture and then the extensions required in order to support network-wide addressing. The naming and coherency schemes are briefly described and we conclude with an evaluation of the approach taken.

2. THE EXISTING MONADS ARCHITECTURE

The MONADS architecture supports a very large virtual memory in which objects are provided with unique addresses which are never re-used. Unlike conventional virtual memories the MONADS virtual memory is persistent. This combination of large unique addresses and persistency means that long-term data and code can be directly stored in the virtual memory, obviating the need for a separate file store.

The virtual memory is decomposed into a collection of contiguous address ranges called *address spaces*. Address spaces are allocated for use as *process stacks*, and as *modules* (see below). An address space number (by which an address space is identified) is never re-used.

The programmer or compiler sees the contents of an address space as a collection of *logical segments*, which are addressed via *segment capabilities*. A segment capability consists of the full virtual address of the start of the segment, the segment length and access rights/type information. Segment capabilities cannot be manufactured or arbitrarily modified by programs, and thus provide the basic memory protection mechanism.

The kernel's memory management routines view an address space as a sequence of fixed length pages, using a model in which page boundaries and segment boundaries are decoupled [10]. Both small and large segments may thus be handled efficiently, without creating the severe internal fragmentation usually associated with combined paging and segmentation schemes [19, 21]. Segments are mapped onto the paged virtual address space via the segment capabilities as shown in figure 1.

The MONADS architecture provides direct support for information-hiding modules. A *module* consists of some private data and routines for accessing that data [11]. All of the data segments belonging to a particular module are held in a single address space. In order to call an interface procedure of another module the executing module must present a *module capability*, which contains a unique module number, an access rights field (indicating which of the module's interface procedures can be invoked using this capability) and some system indicators. The module number is in fact the address space number of the address space containing the encapsulated segments. Each module address space contains some red-tape information from which the location of the code procedures for that module type can be obtained. A user application typically consists of a number of interacting modules.

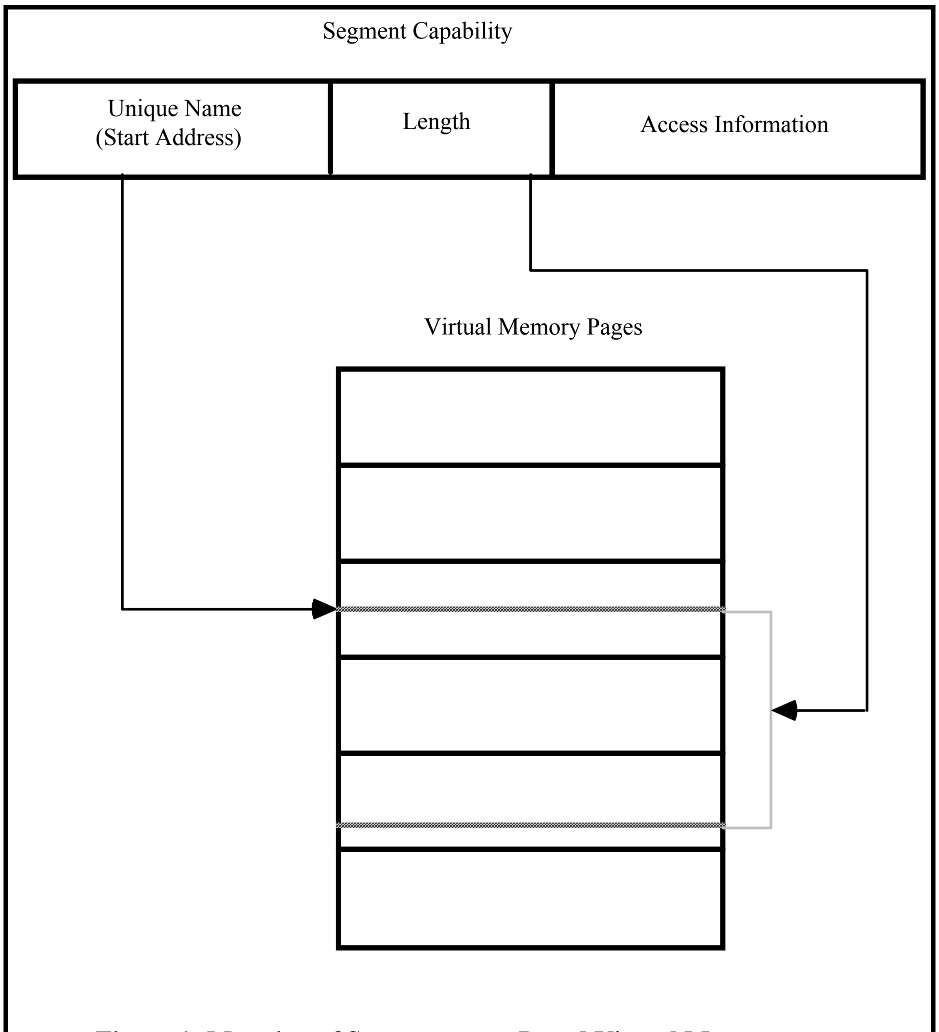

Figure 1: Mapping of Segments onto Paged Virtual Memory

Each MONADS node has an associated *address translation unit (ATU)* [1] which maps virtual memory addresses onto physical memory addresses. This ATU is effectively an inverted page table implemented as a hash table with embedded overflow (in hardware) and its size is thus proportional to the size of physical memory. Hence the use of large virtual addresses does not greatly affect the speed of translation. The structure of a virtual address is shown in figure 2.

The complete addressing process is therefore as follows. A segment capability is used to address a particular segment. The virtual address from this segment capability is added to an offset from the instruction to generate the virtual address of the data to be accessed. The page address portion of the virtual address is hashed and looked up in the ATU. If a match is found then the physical memory address of the required page is obtained from the ATU and the access may proceed. This process is illustrated in figure 3. Otherwise a page fault interrupt occurs and control is transferred to the MONADS kernel. The kernel loads the required page into memory from disk, updates the ATU hash table and retries the instruction. In addition to the access rights checking included at the segment capability level, the ATU also supports read-only pages and causes an exception if an attempt is made to write to a read-only page.

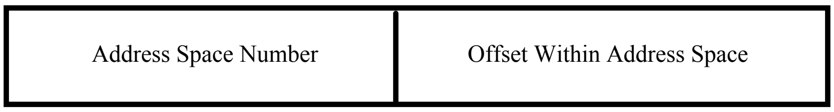

Figure 2: A Virtual Address (Existing Architecture)

A MONADS node may have several disk drives. To facilitate the location of pages on disk, all the pages of an address space reside on the same disk. The address space number is sub-divided into two fields called the *volume number* and the *within volume address space number*. The volume number is the logical disk number of the disk on which the address space is stored. The logical disk number of each disk is written into a well-defined location within the first block of each disk so that the disk may be identified on power-up. The kernel dynamically maintains a table mapping physical disk drives to logical disk numbers. The within volume address space number is unique (forever) for that volume. Each volume contains a *volume directory* in which the address spaces stored on that volume (and the location of their first page on disk) are defined. The disk page table for each address space is stored in the address space itself. Thus, on a page fault, the volume number field of the violating virtual address will identify the disk on which the page resides, the volume directory will indicate the disk address of the first page of the address space and, using this, the disk address of the required page may be located.

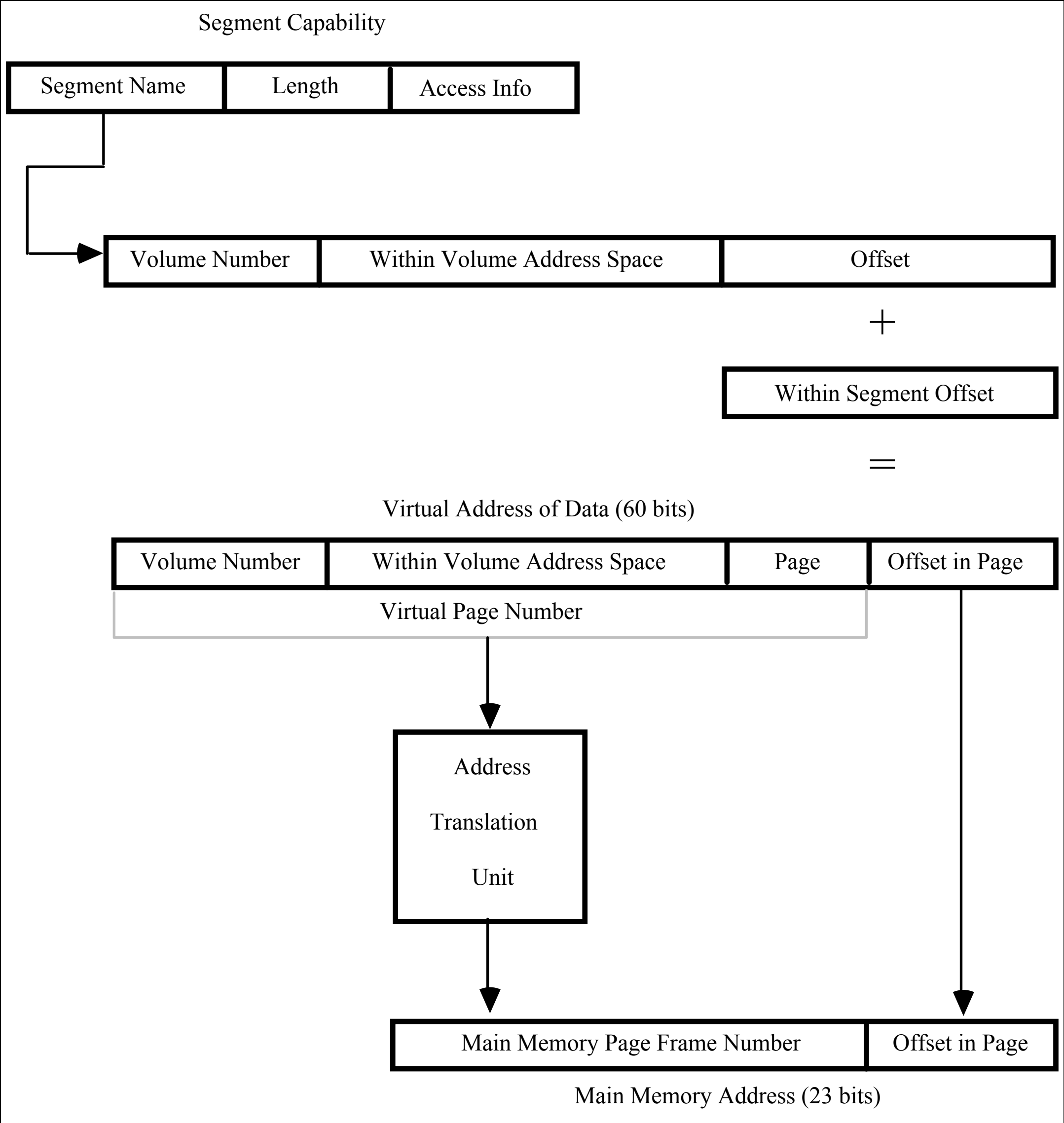

Figure 3: MONADS-PC Address Translation

The MONADS-PC implementation of the architecture has 60-bit virtual addresses, consisting of a 32-bit address space number and a 28-bit offset within address space. The latter is further decomposed into a 16-bit within address space page number and a 12-bit offset within page. MONADS-PC pages are thus 4K bytes long. Address space numbers are subdivided into a 6-bit volume number and a 26-bit within volume address space number as in figure 4[1]. We are currently involved in a new implementation of the MONADS architecture known as the MONADS-MM which will support, amongst other things, a very large physical memory (in the order of gigabytes) [25].

Volume Number (6 bits)	Within Volume Address Space Number (26 bits)	Within AS Page (16 bits)	Offset (12 bits)

Figure 4: Full Structure Of A Virtual Address (Existing Architecture)

[1] The size of some of these fields, particularly the volume number, is clearly too small for a realistic system. MONADS-PC was always viewed as a prototype implementation. The MONADS-MM, a new implementation of the architecture, will have 128 bit virtual addresses, a 64 bit volume number, a 32 bit within volume address space number and a 32 bit offset within address space.

3. THE NETWORK EXTENSIONS

The addressing scheme described above provides a large single-level store for a single MONADS-PC system. The approach taken in the network implementation is to extend this virtual store to encompass the entire network. This is achieved in a completely transparent fashion, so that any byte within the entire network may be directly and uniquely addressed from any machine in the network.

In the following discussion we are concerned with how the virtual store is extended, and therefore issues concerning the actual physical connections between machines are ignored. We simply assume that it is possible to transfer data between any two machines[2].

3.1. Node Numbers

To achieve uniqueness in naming across the network, a *node number* is allocated to each MONADS system[3] . Every node on a network (and in the world) has a unique node number and each logical disk on a node has a unique number within that node[4]. A full virtual address now consists of a unique node number, a volume number within node, an address space number within volume and an offset within address space. This structure is shown in figure 5. When a new address space is created it is allocated an address space number with the unique node number of the creator embedded within it. Thus every address space number, and therefore module number, is unique network-wide.

Node Number	Volume Number	Within Volume Address Space Number	Within AS Page	Offset

Figure 5: Structure Of A Virtual Address (Network Architecture)

The algorithm for resolving a page fault is extended as follows. On a page fault the kernel inspects the node number of the virtual address. If it is the local node number then the resolution proceeds as described earlier and the page is retrieved from a local disk. On the other hand, if the node number is not the local one then, subject to the discussion in section 3.3, the page fault is considered to be a remote page fault. A message is transmitted over the network to the node specified in the virtual address, requesting that a copy of the corresponding page be sent. To enable this, the kernel maintains a table mapping node numbers to network addresses for nodes in a similar manner to the ARP/RARP protocol [6]. The processing of the page fault in the local node is identical once the page has been received, regardless of whether it came from a local disk or a remote node.

Each node with disks must provide a page server facility. Apart from the coherency issues discussed in section 3.4, this involves little extension to the existing local page fault resolution mechanism. On receipt of a request for a page the node determines whether the requested page is already in physical memory. If it is not, then it must be paged in. This is achieved using the existing mechanisms by simulating a page fault on the required page. The page is then transmitted to the requesting node. When the page is received by the requesting

---

[2] In the prototype implementation all machines are connected to an ethernet.

[3] For the modified MONADS-PC, the volume number field of a virtual address is divided into two fields called the *node number* and the *within node volume number*. The node number is 2 bits and the within node volume number is 4 bits.

[4] The fields of a virtual address on MONADS-MM are large enough to ensure that this is possible. Node numbers are 32 bits and within node volume numbers are 32 bits.

node, it is mapped into the local ATU and the waiting process(es) reactivated as for a local page fault.

3.2 Naming Transparency

In order to achieve naming transparency, we take advantage of the fact that on a MONADS system access to all modules is via capabilities. A capability contains, amongst other things, the virtual address of the address space containing the segments belonging to that module. Since virtual addresses are unique forever, a segment thus has a unique name. This eliminates many of the naming problems traditionally associated with networks [20] since every byte stored in the network has a unique name and will only ever be referenced via that name. In this sense naming is a non-issue in the MONADS style of network.

3.3. Location Transparency

The addressing scheme discussed in section 3.1 describes how an address space number is allocated when a module is created and how the segments of the module can be located in the simple case that it continues to be mounted at the node where it was created. It is clear, however, that a realistic scheme must allow modules to be moved between disks and disks to be mounted at different nodes. In such circumstances the virtual addresses of modules are not changed, but from the viewpoint of module location the node number and within node volume number of a virtual address are treated as *advisory*. At each node the kernel holds appropriate further tables to assist in the location of modules which have been relocated. Since the expectation is that most modules are not relocated, these tables hold only exception information. This is in contrast with the IVY system where each node must maintain a complete copy of the page table for the shared virtual store [16].

The *local mount table* maps between a unique volume number (node number concatenated with the within node volume number) and a physical drive mounted at a node. For disks created by the node each entry in this table has the same node number as the local node, but the table allows disks created at other nodes to be mounted.

Each node maintains a *foreign mount table* which allows remote volumes which have been moved to be found. This table maps between relocated unique volume numbers and the nodes at which they are currently mounted. Both the local and foreign mount tables are maintained dynamically on a need to know basis via a protocol of messages over the network. In order to locate a volume the page fault code checks the local mount table and then the foreign mount table, and then it uses the node number in the virtual address. If the volume has been moved from that node, the reply takes the form of an advisory message, enabling the requesting node to update its foreign mount table and to redirect the page request. A failure indicates that the disk is not on line anywhere in the network.

Since access to a module is controlled by a module capability, and since several copies of a capability can exist in a MONADS network, it is imperative that relocation of a module (as distinct from the relocation of the volume on which the module is stored) leaves the module's capability (and hence name) unchanged. To allow moving of an individual module from one volume to another without changing the module's name, a combination of a *forwarding address* on the module's old volume, an *advisory field* in the module capability, and a *moved object table* is used. This information allows any page of a relocated module to be found without the need for broadcast messages, as used in IVY [16] to improve the efficiency of page retrieval.

The issues discussed in this section are more fully described in [3].

3.4. Coherency

At any time there may be several copies of a particular page in the memory of different machines. This creates a coherency problem which is similar to that of cache coherency in multiprocessors [7, 15, 17]. Consequently similar techniques may be applied. The following is a brief description of the protocol adopted. A complete description appears in [2].

As mentioned earlier, pages in physical memory may be marked as read-only or read/write in the ATU. The default state for a page read from a local disk is read-write while the default state for a remote page is read-only. Any number of read-only copies of a page are allowed to exist in the physical memories of nodes in the network at any one time. The coherency protocol guarantees that at any time there are either (a) zero or more read-only copies of the page or (b) exactly one read-write copy. Thus, if a read-write copy of a page exists in the physical memory of a node, then it is the only copy of the page in physical memory of any node in the network. The kernel of the node at which a page is stored on disk (the "owner node") maintains a record of any copies of a page that have been sent to other nodes in much the same way as page owners in IVY [16] keep a copy set for each page. Unlike the "shares" approach proposed in [9], which allows any node to distribute copies of a "granule" (subject to its share status), a MONADS owner node is the only node allowed to provide a copy of a page to resolve a page fault at another node.

When a request for a read-only copy of a page is made to the owner node, one of several scenarios may apply:

1. no copy of the page exists in main memory network-wide,

2. one or more read-only copies of the page exist in main memory network-wide, or

3. a read/write copy of the page exists in the main memory of a node.

These situations are handled respectively as follows:

1. a copy of the page is obtained from disk on the owner node, and then the copy is transmitted to the requesting node (if the request is remote) with read-only access rights;

2. if a copy of the page exists in the physical memory of the owner node, it is transmitted to the requesting node, otherwise a copy of the page is brought into the memory of the owner node and then the page is transmitted with read-only access rights;

3. the node with the read/write copy is requested to mark its copy as read-only, and to transmit to the owner node either an updated copy of the page, or a message indicating that the page has not been modified since the read/write access was granted. The owner node then maintains a record of the requesting node's access, and transmits a read-only copy of the page.

If a node which has a read-only copy of a page wishes to write to the page, the owner node is informed. The owner node then sends a message to all nodes which have read-only copies of the page (except for the node requesting read/write access) requesting that the page be removed from the nodes' memories. When these requests have been completed with the owner sends a message to the requesting node giving permission for the access to be changed to read-write.

As part of the management of virtual memory at a node, page discard may occur. If the page to be removed is not local, either (a) the page is read-only, or unmodified read/write, in which cases a message is sent to the owner node indicating that the page has been removed, allowing the owner node to update its records or (b) the page is modified read/write, in which case a copy of the page is sent to the owner node and the page is not removed until receipt of the page has been confirmed by the owner node.

This protocol guarantees that all nodes see a consistent view of a page. It can result in considerable network traffic in the case that several nodes are constantly modifying the same page, since the page must be sent back to the owner node each time for distribution. However, this is not a particularly likely scenario since, in order to keep the modifications consistent, the processes on the nodes would have to synchronise their activities at a logical level[5]. Coherency protocol messages are only sent to nodes listed by the owner node as having copies of the page, requiring fewer messages than schemes such as "shares" [9], which broadcasts protocol messages, requiring the processing of messages by nodes with no interest in the subject page.

3.5. Network Wide Protection

For the same reason that network naming is a non-issue, protection of data across the network creates no logical difficulties. The architecture does not distinguish between addressing local and remote data and thus the capability-based protection scheme provides control over access to all data. The problem of protection of data whilst in transit between nodes is of course another issue. To solve this we propose to encrypt data, with encryption being performed directly by the network hardware interface during transmission to reduce overheads.

3.6. Shutdown of a Node

At any particular time the memory of each node contains an essentially random set of pages which may belong to an arbitrary set of nodes. It is important for system integrity that, before a node is shut down, any pages which have been modified are written back to disk. In addition, remote nodes must be notified so that the coherency control data described above may be updated appropriately.

During system shutdown the kernel scans all pages of memory and determines, from information held in the ATU, which pages have been modified. Modified pages belonging to a local disk are simply written back, while modified pages belonging to remote nodes must be transmitted to the owner node. Shutdown does not complete until acknowledgement of receipt is received from all remote nodes involved. If the node being shut down has a read-only copy of a remote page, it notifies the owner node that it is shutting down so that the owner node can update its coherency information.

The node must also retrieve an up to date copy of any of its pages that currently exist with read/write access in the physical memories of remote nodes and any nodes containing copies of pages with read-only access must be instructed to remove them from their memory. Access to a removed page by a process at the remote node results in an exception condition.

3.7. Crash of a Node

[5] In any case the effects of such a situation on network traffic can be considerably reduced (but not entirely eliminated) by introducing a small delay before removing write access from a page.

As was described in the previous section it is critical for the correct operation of the system that, when a node goes off-line, it performs the correct shutdown sequence. Unfortunately this may not be possible in the case of a system crash at a node. Such a crash may be due to hardware or software failure and in both cases it is unlikely that a normal shutdown will be possible. Within a single node the consistency and integrity of the store can be guaranteed by using shadow paging [18], as described in [4, 26, 27], to implement a stable store.

In [23] we describe how to implement a stable store on a single MONADS-PC. Each volume is moved from one stable state to the next by a sequence of checkpoint operations. Checkpoint operations may be instigated by a system call from a user program, or automatically by the kernel, for example to free up disk space occupied by shadow pages. A Shadowed Pages Table (SPT) is maintained for each volume to detect which pages from the volume have been shadowed since the last checkpoint operation. Stability is achieved on a per volume basis, and there is a multi-volume stabilise to ensure that cross-references between volumes do not cause inconsistencies following a crash. Only modified pages are shadowed, and at most two copies of a page exist on disk at any time, the latest version and the version as at the last checkpoint. At each checkpoint, reference to the SPT allows the disk space used to store the previous stable version of modified pages to be returned to the free disk space pool. The new stable version of pages form the basis for further system operation.

As described earlier, each MONADS volume contains a volume directory, which exists *in virtual memory*. The root page of the volume is effectively the root of a tree of disk addresses of pages on the volume, and from it the disk location of any page on the volume can be located. After a checkpoint operation, a new tree is constructed, leaving the tree representing the last stable state intact. The last stage of a stabilise operation on a volume is the writing of the root page of the tree to one of two possible locations using Challis' algorithm [5] to ensure that the write operation is atomic. Both of the root pages are placed at well known disk locations so that they can be located and compared on system startup. Thus, following a crash, the store returns to the last consistent state.

In the network environment there are several new problems. These include:

- a request for a read/write copy of a page being denied because a "crashed" node has not signalled its removal of the page from its page table,

- requests for read-only access to a page being denied because a read/write copy of a page exists on a non-responding node, and

- a node is unable to complete its shutdown sequence because it cannot return modified pages to a node which has crashed.

These problems are serious because they can result in a deadlock situation in addition to the loss of data integrity. In [8] we describe extensions and modifications to the single node stability scheme that ensure stability in a network of MONADS-PC computers.

4. EVALUATION

Implementation of the network involved changes to the page-fault handler and creation of a "black box" process which looks like a disk to the page-fault handler.

Modifications to the page-fault handler enable it to (a) detect local and remote faults and to request pages from local disks or remote nodes as necessary, (b) provide a page server function for remote nodes, (c) maintain tables as described above, (d) maintain data coherency, (e) take appropriate action at system shutdown, and (f) maintain system stability.

The new "network" kernel process accepts page requests from the page-fault handler in exactly the same way as does the disk process. It hides network details (such as the underlying network architecture) from the rest of the MONADS kernel by (a) maintaining information about physical addresses and which nodes are currently on line, and (b) providing pages to the page-fault handler by requesting them from the appropriate remote node.

The network process supports two types of message, short messages for page requests and protocol implementation, and long messages for page transfer. Page transfers are 4K bytes each. For small (less than 4K byte) objects, the unit of transfer is therefore larger than the object size, which means that the quantity of data transmitted between nodes is more than immediately required. Many applications exhibit locality of reference, in which case some of the apparently superflous data will in fact be needed in subsequent memory accesses[6]. For objects larger than one MONADS page, the described system is more efficient than systems that move the whole object to a remote node for processing at that node since only the pages of the object that are being processed are transferred.

Application software written to run on a non-networked MONADS processor runs without change on a network of MONADS nodes in a stable virtual store. This is possible because, as explained earlier, pages of data are transferred in order to resolve page faults rather than in response to network specific commands.

5. CONCLUSION

Traditionally the implementation of a network involves major changes to the system software and application software and the provision of large and complex utility software. In this paper we have described an alternative structure based on a network-wide uniform virtual memory. Security and protection are achieved via the use of capability-based addressing and coherency is guaranteed by the paging protocol. The store we have created exhibits stability, so that consistency of data following node or network failure is guaranteed. A major advantage of the scheme is that it involves little modification to the system software and is fully transparent to application software.

A trial implementation of the network is currently being completed. The trial system uses ethernet as the physical medium and will have three MONADS-PC computers as nodes. Our experiences with this system will be reported at a future date.

ACKNOWLEDGEMENTS

The MONADS project is supported by grants from the Australian Research Council and the Newcastle University Senate Research Committee. Thanks are also due to Dr. D. Abramson and Mr. P. Brössler for their earlier contributions and suggestions.

REFERENCES

1. Abramson, D. A. "Hardware Management of a Large Virtual Memory", *Proc. 4th Australian Computer Science Conference*, Brisbane, pp. 1-13, 1981.

[6] Conventional paged virtual memories also rely on this locality in order to achieve acceptable performance.

2. 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.

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*, Amsterdam, pp. 105-116, 1987.

4. 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.

5. Challis, M. F. "Database Consistency and Integrity in a Multi-user Environment", *Databases: Improving Usability and Responsiveness*, ed B. Scheiderman, Academic Press, pp. 245-270, 1978.

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

7. 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.

8. 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*, Bremen, West Germany, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and British Computer Society, pp. 246-256, 1990.

9. Hsu, M. and Tam, V. "Managing Databases in Distributed Virtual Memory", Harvard University, Technical Report TR-07-88, 1988.

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

11. Keedy, J. L. "The MONADS View of Software Modules", *Proc., 9th Australian Computer Conference*, Hobart, pp. 560-574, 1982.

12. Keedy, J. L. "A Memory Architecture for Object-Oriented Systems", *Objekt-orientierte Software und Hardwarearchitekturen*, ed H. Stoyan and H. Wedekind, Teubner-Verlag, Stuttgard, pp. 238-250, 1983.

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

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

15. Knapp, V. and Baer, J.-L. "Virtually Addressed Caches for Multiprogramming and Multiprocessor Environments", *Proc., 18th Hawaii International Conference on System Sciences*, pp. 477-486, 1985.

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

17. Linn, C. and Linn, J. "The Carrick-on-Shannon Architecture: A two-level Cache-Coupled Multiprocessor Architecture", *Proc., 18th Hawaii International Conference on System Sciences*, pp. 487-504, 1985.

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

19. Organick, E. I. "The Multics System: An Examination of its Structure", MIT Press, Cambridge, Mass., 1972.

20. Popek, J. and Walker, B. "The LOCUS Distributed System Architecture", The MIT Press Series on Computer Systems, pp. 14-17, 40-46, 1985.

21. Randell, B. "A Note on Storage Fragmentation and Program Segmentation", *Communications of the ACM*, 12, 7, pp. 365-369, 1969.

22. 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.

23. 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*, Bremen, West Germany, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and British Computer Society, pp. 229-245, 1990.

24. Rosenberg, J. and Keedy, J. L. "Object Management and Addressing in the MONADS Architecture", *Proceedings of the International Workshop on Persistent Object Systems*, Appin, Scotland, 1987.

25. Rosenberg, J., Koch, D. M. and Keedy, J. L. "A Massive Memory Supercomputer", *Proc. 22nd Hawaii International Conference on System Sciences*, vol 1, pp. 338-345, 1989.

26. Ross, D. M. "Virtual Files: A Framework for Experimental Design", University of Edinburgh, CST-26-83, 1983.

27. Thatte, S. M. "Persistent Memory", *Proc. IEEE Workshop on Object-Oriented DBMS*, pp. 148-159, 1986.