Distributed Persistent Stores

Frans A. Henskens and John Rosenberg

Basser Department of Computer Science
The University of Sydney
N.S.W. 2006
Australia
Email: {frans, johnr}@cs.su.oz.au

Abstract

Considerable effort has been expended over the last decade on the production of persistent stores, for example CMS POMS, CPOMS, MONADS and Thatte's Persistent Memory. These generally did not provide for concurrent access to the store, and included no support for distribution. In this paper we investigate the distribution of persistent stores, including the issues of uniformity, unbounded size, and stability. We then examine and contrast two examples of distributed persistent stores, the Monads DSM and distributed Napier.

1 Introduction

Persistent systems support mechanisms which allow programs to create and manipulate arbitrary data structures which outlive the execution of the program which created them [4]. This has many advantages from both a software engineering and an efficiency viewpoint. In particular it removes the necessity for the programmer to flatten data structures in order to store them permanently. Such code for the conversion of data between an internal and external format has been claimed to typically constitute approximately thirty percent of most application systems [4]. In this sense a persistent system provides an alternative to a conventional file system for the storage of permanent data. This alternative is far more flexible in that both the data and its interrelationships can be stored in their original form. In order to achieve this a uniform storage abstraction is required. Such an abstraction is often called a *persistent store*. A persistent store supports mechanisms for the storage and retrieval of objects and their interrelationships in a uniform manner regardless of their lifetime.

Whilst considerable effort over a number of years has been expended in the production of persistent stores, for example CMS [6], POMS [14], CPOMS [10], MONADS [38], and Thatte's Persistent Memory [45], these included no support for distribution. Distribution of a store across separate machines allows users of these machines to share the resources contained in the store. For example programs and data become available to all users (subject to protection issues), and facilities such as electronic mail become implementable. Note that this discussion has assumed that multiple users are able to simultaneously access the store. Most of the above mentioned persistent store implementations do not support such concurrent access.

Distribution requires connection of the machines using a medium which allows them to communicate. Work to date on distributed persistent stores has focussed on machines connected to form a Local Area Network (LAN) [23, 48]. LANs are formed by connecting the processors or *nodes* to a contention bus such as Ethernet, or by point to point links to form a ring which is managed using protocols such as Token Ring [20], Slotted Ring [37] or Cambridge Ring [35]. The LAN technology is highly developed, and provides a means of

Published in: *Journal of Microprocessors and Microsystems*, 17(3), Butterworth-Heinemann, Oxford, U.K., pp. 147-159, 1993.

reliably transferring messages between nodes at a speed similar to the transfer of data from a disk.

Conventional systems typically utilise the message passing facilities provided by LANs through various higher level protocols. The widely used TCP/IP protocol suite [15], for instance, provides a virtual network, hiding the architecture of the LAN from the user and providing a consistent interface to the network to users of different LAN types. Using these protocols processes executing on separate computers are able to communicate. Such communication is called interprocess communication (IPC). IPC may be achieved using a number of different models, including remote procedure call (RPC) [49] and distributed shared memory (DSM) [44].

RPC allows a process to invoke a procedure running on a remote processor, and to transmit and receive data back in the form of parameters to the procedure call. DSM provides the abstraction of a shared physical memory to processes running on processors which are not tightly coupled. It has the advantage that processes can share data structures such as arrays and linked lists as well as the flat data that can be logically shared using RPC. Implementation of both RPC and DSM requires the existence of an underlying message passing system.

In this paper we discuss issues associated with the implementation of distributed persistent stores, and examine and contrast two examples, distributed Napier [32] which uses RPC at the operating system level to implement a persistent DSM, and the Monads DSM [26] which provides a persistent DSM at the architectural and operating system level.

2 Distributed Persistent Stores

Persistent stores provide for the storage and manipulation of data objects by persistent programming languages. According to [11, 16] properties required of such a store include:

(1) Uniformity. A single mechanism is used for storage of all types of data.

(2) Unbounded size. The physical properties of the storage media are hidden, and the store provides the abstraction of having infinite size.

(3) Stability. All details of the movement of data between main and secondary storage is abstracted over. Failures are hidden, resulting in a store that appears to be error free.

Two distinct approaches to the construction of such stores have emerged:

(1) the building of a layered architecture on top of an existing hardware platform and operating system (for example [4, 9, 10, 11, 45]), and

(2) the construction from scratch of dedicated architectures such as the Monads architecture [31].

The principle underlying the implementation of most persistent stores is *persistence by reachability*. According to this principle. the fact that an object A refers to another object B implies that B will persist as long as A does [21]. This implies that an object exists in the persistent store if it is referred to by another object in the persistent store. A further implication is that there is a persistent object whose existence is not dependent on its being referenced by another persistent object. This object is called the *root of persistence* for the store. The root of persistence is maintained at a well known location in the store, and points to a graph structure containing all the persistent objects in the store. Data objects are created by programs. Such data may be either transient data (for example local variables) which does not persist beyond the life of the program, or persistent data. Persistent data may include databases, libraries, or the program code produced by compilers. All persistent data objects are reachable from the root of persistence.

Distribution of a persistent store entails more than simply connecting the host machines to a network. Satisfaction of the uniformity requirement, for instance, precludes any difference in the user perception of access to local or remote objects. The store itself must be able to detect the need for use of the network in accessing remote objects. The unbounded size property poses few problems not experienced in the single node case. In fact distribution of the store could be seen to imply fragmentation of the secondary store across separate nodes, which suggests that the physical size of persistent stores may benefit from distribution. In a logical sense, however, single node or distributed stores should both present the abstraction of unbounded size. The stability requirement necessitates the ability to move the store from one stable state to another as an atomic operation which may involve write operations on disks connected to separate nodes. This implies the existence of a coordinator for the stabilise operation. The stability abstraction additionally involves coping with the possibility of network failures. If the store is to be used simultaneously by multiple processes, there is also a requirement for *concurrency control*. That is, there must be a method of allowing a set of processes to interleave a set of operations on the store in a controlled and predictable way. Another implication of concurrent access to the store is the issue of *protection*. If all processes execute within the same store, it is necessary to provide a means by which users can restrict access to their data.

We examine these issues in more detail below.

2.1 Uniformity

The uniformity property requires use of the same mechanisms for storage of long-term and transient data of any type (for example, graphical image, procedure or integer). A crucial issue here is that of *naming*. Access to objects is achieved through the ability to identify or name them. For example the single-user Napier88 system [11] uses Persistent Identifiers or PIDs. If the store is distributed, users must be able to access locally stored and remotely stored objects in the same way. This implies that the persistent store is able to transparently detect whether an object is available locally, or if it is necessary to obtain access to the object using the network. In the case that the object is remote, the location of the object must be determined. If the object store is centralised, as with Distributed Napier [32], this is trivial because all remote objects are obtained from the same source. If, however, the object serving function is itself distributed (as in Monads DSM [23]) the local store manager must be capable of identifying the appropriate server prior to requesting the object. This is a non-trivial issue, particularly if objects are allowed to migrate between servers during their life [24].

Location of distributed objects may be achieved in a number of ways:

(1) *Use of broadcast messages when requesting access to such objects*. These messages must be processed by all nodes, and this represents a significant overhead for the nodes with no knowledge of the required object. We reject this technique because of its inefficiency.

(2) *Use of a centralised object server*. This technique uses a special node called the *central server* which maintains the only copy of shared data, and which provides blocks of data on request and stores modified blocks. At any instant in time there is at most one reader or one writer of any block. Extension of the central server protocol to incorporate *read replication* is not difficult. This extension provides for either multiple simultaneous readers of a data block or a single writer. The central server with read replication model, which is implemented in Distributed Napier, renders access to objects reliant on the up/down status of the server. In addition bottlenecks can occur at the server at times of heavy access. These problems can be

lessened by distributing the object server function using an object server group (eg Choices DVM [27]), each member of which provides a subset of the objects. Requests are made of the group, and the server which is able to do so satisfies the request. The use of server groups is most effective if the underlying network supports multicast messages [1] which allow object requests to be transmitted only to the server nodes.

(3) *Use of distributed object servers coupled with structured object names*. The requesting node is able to identify the appropriate object server from the object name and direct the object request to the server. This technique results in better resilience to server failure and also reduces the incidence of server bottlenecks. The problem is that if objects are moved between servers a corresponding change of name is required to reflect the identity of the new server. A solution (eg V [13]) is to remove the name handling function from object servers. The identity of a name server group is used as a prefix to object names, and the identity of the current server for an object is obtained from the name server. This technique provides increased flexibility at the expense of increased network traffic. An alternate method of providing for movement of objects between servers combines the inclusion of advisory information in object names and forwarding information at servers as a means of locating moved objects. This technique is implemented in the Monads DSM.

2.2 Unbounded Size

Since no store can in fact have the property of unbounded size, an *illusion* of unbounded size is provided. In order to maintain this illusion using a store of finite physical size, unreferenced objects must be removed from the store, thus freeing the physical space occupied by the objects. This process is called *garbage collection*. Garbage collection is the act of removing objects that are no longer needed by the user(s) of the store. Such objects include

(1) transient objects such as stack frames and other temporary data structures created by a program and no longer needed by it, and

(2) objects which had been placed in the persistent store, are no longer needed by the owner, and are therefore no longer referenced.

Garbage collection of an entire persistent store is an expensive process because it involves computing the transitive closure of all objects reachable from the root of the store. Any objects not included in this transitive closure are garbage and may be removed. For a distributed store with multiple object servers this involves following pointers across nodes, resulting in network messages dedicated to garbage collection. To reduce the expense of garbage collection some persistent stores create and modify objects in a segregated area called a local heap [9] which may be frequently garbage collected in isolation from the main store.

A common technique for garbage collection is partitioning of the store into semi-spaces and implementation of a compacting garbage collector which copies referenced objects from one semi-space to the other. Objects remaining after a complete copy are garbage and may be overwritten by the next compaction (eg [48]). An improvement on this technique is the generation garbage collector [47] which partitions objects according to their age and garbage collects the youngest partitions more often than the oldest ones. This technique relies on the fact that younger objects are most likely to be transient.

2.3 Stability

When a computer system shuts down unexpectedly due to hardware or software failure the contents of volatile memory or RAM is typically lost, whereas data stored in non-volatile memory such as disk or tape usually remains available on system restart. Since accessing RAM is much faster than accessing disk or tape, processors typically manipulate data in RAM. Persistent stores provide for storage and retrieval of data and its interrelationships in a uniform manner that is totally independent of the lifetime of the data. Uniformity of storage and retrieval of data is achieved by hiding the separation between disk and RAM storage. The transfer of objects and/or parts of objects between the disk store and RAM, and indeed between nodes of a distributed store, is controlled by the store in a manner which is totally transparent to the user.

At any time the true picture of the state of a persistent store is a combination of the data currently held in RAM and other data stored on disk. It is crucial to effective system management that the integrity of the store be guaranteed, especially after an occurrence such as a system crash or hardware failure.

Failure of the store may result in

(1) the *destruction* of data caused by, for instance, the disintegration of a disk platter or a head crash[1], or

(2) the *corruption* of data, due to an unexpected system shutdown which causes the data to become inconsistent.

A store is deemed to have the required level of integrity if, following a failure, it always reverts back to a consistent state that existed prior to the failure. Persistent systems provide such integrity, appearing to the user to be failure free. A store which provides the illusion of being failure free is said to be *stable*.

A number of proposals for stable stores have appeared in the literature (eg [5, 22, 42, 45, 46]), and many of these are based on a scheme proposed by Lorie [34] called *shadow paging*. This technique, which was proposed for database systems, steps the database from one stable state to the next using a process called *checkpointing*. The database is divided into fixed sized units called *pages*, and between checkpoints, two versions of modified pages of the database are maintained. These are:

(1) the version of the page at the last checkpoint, called the *shadow page*, and

(2) the *current page*, which is the shadow page plus all modifications made since the last checkpoint.

At each checkpoint the current pages are written to disk and become the shadow pages, and the previous shadow pages are reclaimed as free disk pages. If a system failure occurs the system reverts to the state at the last checkpoint. This state is represented by the set of unmodified pages plus the set of shadow pages.

There are two techniques for the creation of shadow pages called the *before-look* and *after-look* strategies. According to the before-look strategy a disk copy (shadow) of a page is taken before any modifications are made to it, and the current version of the page (which includes modifications made since the last checkpoint) is written to the original disk location. Reversion to the previous stable state involves overwriting these current versions with the shadow versions. This scheme has the advantages that the disk page table remains static

---

[1] Physical failure of the storage media is best handled by a dumping or backup strategy, and is not discussed further here.

except for newly created pages (a table of the disk positions of shadow pages must be maintained between checkpoints), and the disk position of pages is retained (preserving clustering of pages as required by some disk performance enhancement schemes).

The after-look strategy treats the disk copy of a page after a checkpoint as the shadow page, and writes modified (current version) pages to a new disk location. This scheme has the advantages that a page copy does not occur prior to the first post-checkpoint modification of a page (because the shadow copy already exists), and that the checkpoint-modify-checkpoint cycle requires one less disk write per page than the before-look strategy. The scheme does not, however, maintain clustering of pages on disk, and requires dynamic updating of the disk page table to reflect modified and newly created pages.

A checkpoint causes a transition of the store to the next stable state, and this transition involves a series of disk write operations. It is important for the integrity of the store that this series of operations is seen as an atomic unit which either occurs or fails. The atomic update feature provides such an abstraction, making it impossible for the system to be left part way through a checkpoint operation.

A suitable technique for achieving the effect of an atomic update is the use of two roots for the store. One of these roots points to the most recent stable state of the store and the other points to the previous stable state. The last step of a checkpoint operation involves overwriting of the root pointing to the previous stable state with a new root. This new root points to the stable state being created by the checkpoint operation. The actual storing on disk of the root for the next stable state must itself be achieved as an atomic operation.

It is possible to verify the correctness or atomicity of a disk block write operation by structuring the block with identical time stamps as the first and last words. This technique was first described by Challis [12]. When determining the correctness of such a disk block these time stamps are read, and if they are identical then the last write operation on the block completed correctly. By maintaining two such blocks, storing the roots for the last two checkpoint operations, in well known locations on disk, and by only overwriting the older or the incorrect block, it is possible to use these blocks to point to the last stable state of the store [39].

Conceptually it is possible to checkpoint all of a persistent store at once. This requires that all processes executing within the store cease execution during the checkpoint operation. The length of time such an operation would take is a function of the extent of modifications to objects since the last checkpoint operation: this extent is a function of the nature of the processes themselves and of the period of time between checkpoints. In any event it is not acceptable that users are globally affected by checkpoint operations.

Another strategy is to checkpoint *groups* of persistent objects separately, thus allowing some users to continue working while the data used by other user(s) is checkpointed. This technique has the advantages that:

(1) checkpointing may then be controlled by the user as well as by the persistent store, allowing the possible use of checkpointing as a basis for transactions [8], and

(2) checkpointing of the entire store is divided into a number of smaller operations, each of which may complete relatively quickly.

Allocation of the objects in the store to groups may, for instance, be made according to the owner of the objects or their storage location on disk. In either case it is possible that references between groups may create interdependencies which require that they are stabilised together. Modified objects belonging to groups may at the time of the checkpoint be held in object caches at different nodes, requiring that they be retrieved by the object server for the checkpoint operation to proceed. Objects belonging to a group may be managed by different servers. These issues require the design of protocols which can provide the illusion of an atomic write operation involving multiple objects and distributed object servers.

In the following sections we describe the architectures of two distributed persistent stores; the distributed Napier store which implements the centralised server model, and the Monads DSM which implements the distributed server model.

3 The Distributed Napier Store

The Distributed Napier store [32, 48] is constructed above the Mach [3] operating system. Mach provides the implementors with features including:

(1) the ability to provide processes with alternate page fault handlers called *external pagers*,

(2) distributed IPC through a remote procedure call mechanism which allows location independent procedure calls, and

(3) multiple threads of execution in a single process address space.

The store comprises a single four gigabyte paged address space called the *persistent address space*. This store implements a heap of persistent objects which are addressed using their virtual addresses. Thus address translation hardware can be used in object management, with fault conditions being handled by the external pager associated with each client process. This external pager maintains a *local page cache* of pages from the persistent store for its client; a special subset of these pages is termed the process' *local heap*. Persistent objects are paged into the local page cache as required, and newly created objects only are stored in the local heap. Local heap pages are allocated from the persistent store, so addressing the local heap is consistent with addressing the persistent store. The local heap pages are logically special, however, because objects stored in them cannot be shared with other clients. Before an object created by a client can be accessed by other clients it must be copied from the local heap into persistent pages, thus becoming a (potentially) persistent object.

3.1 Distributed Access

Distribution is achieved using a combination of the central server and read replication models [43] (see section 2.1). This combination provides a single page server which transmits copies of pages to client processes as required, and which stores modified pages on discard from local page caches. The server function is performed by the *Stable Store Server*. Coherency of the store is achieved at the page level using a multiple reader/single writer protocol. According to this protocol, either multiple read-only copies of a virtual page may exist in clients' local page caches at the same time, or at most one read-write copy may exist in a local page cache on a system-wide basis. The protocol ensures that processes accessing shared data all have the same view of that data. The Stable Store Server is centralised and consists of:

(1) The *Server Request Handler*, which receives requests from clients in the form of IPC messages and services these requests by passing them to the appropriate components of the server. In handling these requests the Server Request Handler also implements the coherency protocol through its internal *Coherency Manager*.

(2) The *Stable Store Manager*, which reads pages from, and writes pages to the stable medium (the physical store).

(3) The *Stable Heap Manager*, which provides the abstraction of a store consisting of a heap of persistent objects. This manager is called, for instance, when a new persistent object is created.

Each client executes against a subset of the stable store pages which are contained in the local page cache. This cache is maintained by the client's external pager, which communicates with the Stable Store Server using the *Client Request Handler*. Part of the local page cache contains the client's local heap, which consists of a contiguous set of previously unused pages from the persistent address space, and in which the client places newly created objects. The local heap is locked into main memory so that pages from it may not be discarded by the system. Since the local heap contains local objects, its pages are not shared with other clients. A client may make one of its newly created objects persistent by creating a pointer to it from some other persistent object. After a local heap object becomes persistent, another client may attempt to access it by following such a pointer. Such access could only occur if the other client were able to obtain a copy of the persistent page containing the local heap pointer, and would be illegal because local heap pages are private to the client associated with the heap. The strategy which allows the sharing of a persistent page which contains pointer(s) into a local heap is discussed below.

3.2 Naming and Coherency

Objects are accessed by following pointers, and are thus named according to their virtual address. As described below, the virtual address of an object may change during its life as it moves firstly from the local heap to the main heap, and then within the main heap during garbage collection. This changing of the names of objects throughout their life involves considerable pointer manipulation.

Coherency of the persistent store is achieved using a multiple-reader/single-writer protocol applied at the virtual page level. This protocol is controlled by the Coherency Manager section of the Server Request Handler. The coherency protocol moves pages between states according to a finite state automaton. The server request handler maintains the *Export Table*, hashed on page number, to enable coherency control. This table contains an entry for each persistent page currently in a client page cache. Each entry indicates the current state of the page, and two lists. These lists are

(1) The *V-list*, which contains an entry for every client currently holding a copy of the page. A page is always exported with read-only access, and may be provided by any member of the V-list under instruction from the Server Request Handler. When a client requests write access to a page, the other members of the V-list for the page are instructed by the Coherency Manager to invalidate their copies.

(2) The *D-list*, which contains an entry for every client who has seen the page since it was last stabilised. This list is crucial to the stability algorithm because it enables the identification of clients who have accessed the same modified data (see *associations* section 3.3).

A modified persistent page held in a client's page cache may contain a pointer into the client's local heap because the client has made a newly-created object persistent. When another client requests a copy of such a page, or when the client performs a stabilise operation, the local heap object must be moved from the local heap pages into persistent store pages so that other clients may access the newly created persistent object (recall that other clients may not access a client's local heap). Because the local heap page(s) containing the object may also contain non-persistent objects, it is not satisfactory to simply make such page(s) persistent. Doing so would potentially increase the number of transient objects in the persistent store and necessitate more frequent (and costly) garbage collection of the store (recall that the main reason for the maintenance of local heaps is to avoid frequent garbage collection of the persistent store). The solution adopted, and illustrated in figure 1, is to:

(1) Record all persistent pages in the local cache which contain pointers into the local heap in a *remembered set*.

(2) Maintain a set of persistent pages called *copy-out* pages with each client. The maintenance of a sufficiently large pool of copy-out pages is controlled by the coherency algorithm, which ensures that the server is requested to allocate more of them whenever the pool size falls below a minimum level.

Whenever a page in the remembered set is copied from the local page cache to some other client, pointers into the local heap are identified, and the referenced local heap objects are copied into one or more copy-out pages. As the objects are copied, the pointers to them are changed to reflect the objects' new locations. Other pages in the remembered set are also scanned and pointers are changed as necessary. Since copy-out pages may be shared with other clients, this allows such clients to access newly-created persistent objects previously held in a client's local heap. Pointers from the local heap to the moved object(s) are modified using a different technique:

(1) The original local heap pages from which objects were copied are read and write protected, resulting in the generation of a fault condition whenever an object in such a page is accessed.

(2) The local store manager, acting on each fault caused by (1), detects any attempt to access the old copy of an object moved to a copy-out page, and corrects the pointer used for the access so that it points to the copy-out version of the object.

(3) The next garbage collection operation, which necessarily traverses the local heap, detects and corrects any remaining pointers to the old copy of the object.

As a result of (3), the protection on local heap pages may be removed after garbage collection. Use of this technique for modifying local heap pointers avoids the expense of traversing the entire heap whenever a page in the remembered set is exported.

Copy-out pages are also used when the local heap becomes full, and garbage collection is unsuccessful in creating sufficient space for continued operation. The least recently used objects are copied from the local heap into copy-out pages. During this operation some transient objects may be effectively moved into the persistent store, so that when they cease to be of use and become garbage they will not be found during garbage collection of the local heap. The major benefit of the use of local heaps is the ability to localise garbage collection, so it is important that the space allocated to local heaps is sufficiently large to minimise such spill-over of transient objects into the persistent store.

3.3 Stability

Stability is achieved using an *after look* strategy; thus a copy of modifications to an object is made after the modification. In accordance with this strategy, modified persistent pages which are removed from a local page cache as part of page discard or a stabilise operation are written to previously unused *shadow pages* on disk following the technique proposed by Lorie [34]. This means that, temporarily, both the stable and modified versions of the page are stored on disk. The final stage of a stabilise operation updates the disk page mapping information using Challis' algorithm [12], thus creating a new stable state for the store. Prior to this final stage the disk mapping information points to the preserved previous stable state, thus permitting reversion to that state in the case of a client or system failure.

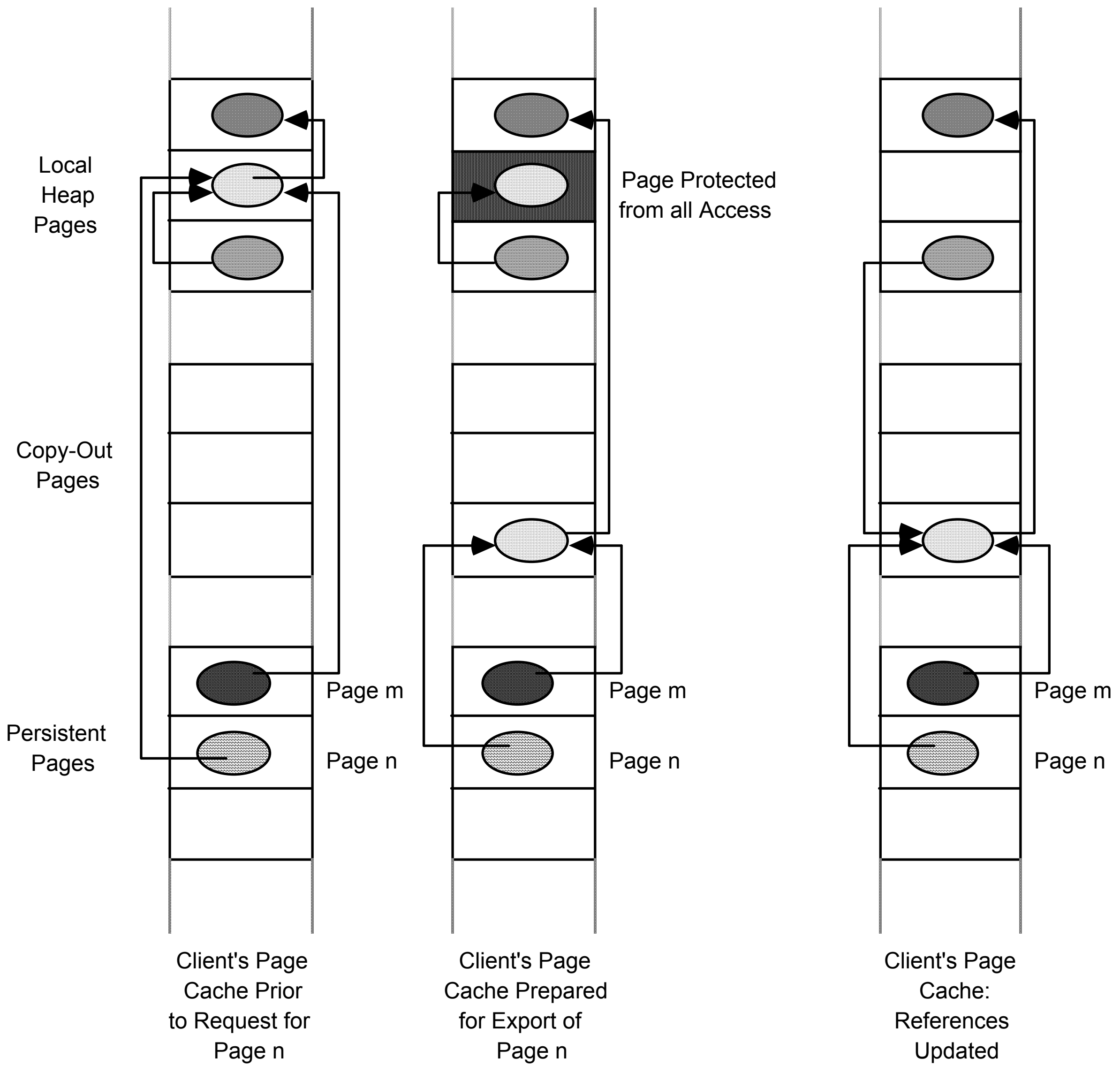
Local Heap Pages
Copy-Out Pages
Persistent Pages
Page Protected from all Access
Page m
Page n
Page m
Page n
Page m
Page n
Client's Page Cache Prior to Request for Page n
Client's Page Cache Prepared for Export of Page n
Client's Page Cache: References Updated

Figure 1. Stages in Exporting a Persistent Page

Checkpoints occur on a per client basis. Because clients may have made decisions based on data which has been modified since the last checkpoint operation, such clients are deemed to belong to dynamic entities called *associations*. An association is a set of clients who have become dependent on each other because they have seen the same modified data. Each association has an accompanying list of persistent pages which contain the modified data which was accessed in common. Associations may change whenever a client obtains a copy of a page that has been modified relative to the stable copy of the page. When a client obtains a copy of a modified page

(1) the association to which the client belongs is *merged* with the association of other clients dependent on the page, and

(2) if necessary, the page is added to the list of pages modified by members of the association since the last stabilise.

When any member of an association requests a stabilise operation,

(1) all members of the association must garbage collect their local heap, and

(2) all pages modified by members of the association must be copied to the stable store as an atomic operation, thus ensuring that the store remains consistent.

3.4 Garbage Collection

Since the incidence of transient objects (which soon become garbage) is much higher amongst newly created objects, the separation between local heap pages and other pages in the persistent store is important, because it allows the local heap to be garbage collected separately and more frequently than the main heap. Local heap garbage collection is carried out on a per process basis, thus minimising its effect on other clients. At present the four gigabyte main store is logically divided into semi spaces and a compacting garbage collector is used. This effectively halves the available address space, reducing to 2 gigabytes the size of store available at any time. The implementors plan to extend the system to encompass two full sized (four gigabyte) address spaces. The garbage collector will compact from one of these address spaces to the other. Processing will switch to the alternate address space on completion of each garbage collection. This would double the available store, making all of the four gigabyte address space available. This scheme has the disadvantage that it requires the suspension of processing during garbage collection of the persistent store.

3.5 Protection

The approach to protection taken by the Napier system is based on strict type security. Code generation is restricted to trusted programs such as compilers, thus ensuring that programs cannot generate illegal addresses. This has some merit in that it shifts most of the protection overheads to compile time. There are, however, major disadvantages in terms of flexibility. In particular, the reliance on types appears to preclude the implementation of mixed language environments.

3.6 Evaluation

The Distributed Napier system effectively provides distributed multi-user access to a persistent store. The use of the central server rather than distributed server model for page distribution increases the possibility of network bottlenecks in the form of page requests, transmissions, and returns, at the server. Whilst it is sensible to couple the page server and coherency management functions, the location of the coherency manager with its associated write request, invalidate, invalidate acknowledge, and change access messages within the stable store server further accentuates the bottleneck problem. In addition, the centralised design results in a system which is totally reliant on the availability of the server. If the server fails, all of the persistent store becomes inaccessible to clients. In a distributed server system, on the other hand, the failure of a server renders only part of the store inaccessible. The design of a distributed server Napier system involves significant extensions to the failure and garbage collection strategies, and is the subject of future research.

Finally, the implementors have reported several problems related to the construction of their store above an existing architecture and operating system [48]. These include lack of operating system support for marking of locally cached pages as unmodified but "precious" (i.e. not to be discarded), deficiencies in exception handling, and a general lack of control over page discard. The solutions to these problems introduce inefficiencies, for example the necessity to non-destructively write to every page brought into the local page cache. These inefficiencies would not occur if a purpose-built architecture and operating system was being used.

4 The Monads DSM

The Monads DSM [23, 26], which provides a distributed persistent store, was constructed above the Monads-PC [38]. This is a purpose-built microcoded computer which provides a 60 bit wide paged virtual memory. The system supports both large objects called *modules* (which are roughly equivalent to files in conventional systems), and small objects called *segments* (containing for instance, procedures, records, characters or integers). Large objects are accessed according to the information-hiding principle [36]. According to this principle a module conceptually consists of data surrounded by interface procedures which provide access to the data. The Monads architecture enforces this mode of access, preventing direct addressing of the data stored in a module. Access to large and small objects is strictly controlled by the use of capabilities [18, 19, 30]. The DSM supports both diskless nodes and nodes with attached disks. Nodes with disks act as servers for the pages of the objects stored on those disks.

The Monads-DSM implements an unconventional approach to memory management. Since this approach is crucial to the DSM's ability to support and transparently address such a large distributed virtual store, it is described in more detail in the next section.

4.1 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. Monads, 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 [40]. The importance of this for distribution is that each Monads node maintains a main memory page table proportional in size to the size of its own *main* memory, and distributes disk page tables across nodes. This is in contrast to other DSM implementations (eg [17, 33]) which 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 partitioned the address space in which a process executed into *local* and *shared* addresses. The shared memory partition only was 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 the Monads architecture allows the entire virtual memory space to be implemented as DSM, distributed across a network of Monads-PC 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 is determining which of the attached disks contains the page.

Every Monads 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 2.

Node No.	Volume Number	Within Volume Address Space Number	Within AS Page	Offset

Figure 2. The Structure of a Monads DSM Virtual Address.

All the pages of an address space are stored on a single disk. 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 Monads 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 Monads virtual memory as a collection of segments which may be of arbitrary size. Segment boundaries are orthogonal to page boundaries [28], thus avoiding internal fragmentation. Segments contain data and capabilities for other segments, allowing complex graph structures to be constructed. Access to a segment is permitted on presentation of a valid *segment capability*. The architecture prevents the arbitrary modification of capabilities. The use of segment capabilities in addressing the virtual memory is illustrated in figure 3.

Segments are grouped together into the modules described above. 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 become available. These 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.

4.2 Distributed Access

The Monads Distributed Shared Memory (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 [2]. Related schemes have been reported in the literature [17, 33]. 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). 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 [15]. 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. This is achieved using techniques similar to those used in virtual memory management.

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 a single large disk.

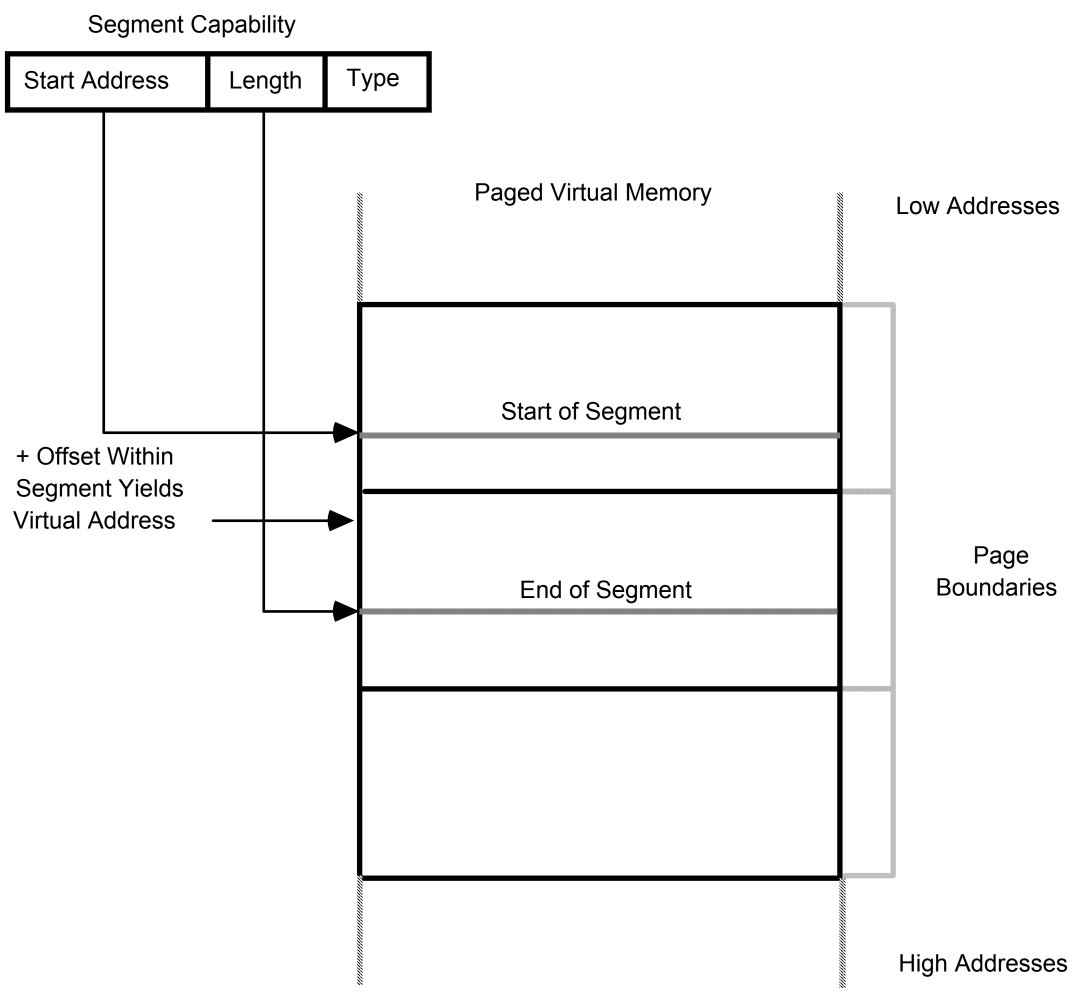

Figure 3. Addressing Using Segment Capabilities.

4.3 Naming and Coherency

As described in section 4.1, 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 4.2, 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 was 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.

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 a local table which maps moved volumes to their new mounting node. If necessary the destination node for the page request message is adjusted accordingly. 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. Subsequent accesses to the pages of such modules occur as if the module had not been moved. This is achieved through the use of an *alias* for the module. Issues and techniques relating to movement of modules are described in more detail in [7, 24].

Coherency is achieved according to a multiple-reader/single-writer protocol on the virtual pages of the store. This protocol is enforced by each server node for the pages of each module stored at that node. The kernel at each server maintains an Exported Pages Table (XPT) listing the importing nodes for each exported page, and the access provided (read-write or read-only) for these pages. Pages are always exported with read-only access, and nodes must request elevation to read-write access if required. Importing nodes are sometimes requested to forward copies of pages to another requesting node. This is more efficient than the page server retrieving a page from local disk in the case that another node has a copy in its local page cache [13], or retrieving and then retransmitting a page in the case that a remote node has modified the page.

4.4 Stability

Stability of the store is achieved on a per volume basis using an after-look shadowing strategy applied to the Monads virtual page. After a stabilise operation or system startup, original data is retained in shadow pages and modifications are made to the copy or *current version* of the pages. The system detects when a page is about to be modified, and if the page has not already been shadowed, a current version disk page is allocated. It is necessary to allocate the current version disk page *prior* to the first modification to ensure that sufficient disk space is available to enable writing the page to disk. If free space is not available, the system stabilises the volume, thus freeing up disk pages allocated to shadow pages for the volume. If stabilising the volume does not result in free disk space, the volume is full, and an exception condition occurs. A stabilise operation may also be initiated by users, possibly as part of higher-level transaction management.

The scheme relies on the following:

(1) The maintenance of a transient *Shadowed Pages Table* (SPT) for each volume. This table indicates, for each disk page, whether the page is allocated to storage of a current or shadow page. It is used to prevent the allocation of duplicate shadow pages for virtual pages, and for the return of shadow pages to the volume free list as part of stabilise operations.

(2) The maintenance *in virtual memory* of disk page tables and disk free space lists so that modifications to these data structures will occur in corresponding current version pages. When a current version page is allocated to a virtual memory page, a corresponding change is made to the disk page table for the address space containing the page. This may involve the allocation of a current version disk page for the virtual page containing the modified disk page table entry.

(3) Support for the marking of main memory pages as read-only or read-write, with a write fault exception condition occurring on an attempt to write to a read-only page. When a write fault exception occurs, the system ascertains whether the page has been

modified since the last stabilise operation. If not, a shadow page is allocated before the modification is permitted to proceed.

(4) The ability to detect whether a main memory copy of a virtual page has been modified since being mapped into main memory. If the page has been modified, it must be written to stable storage as part of the next stabilise operation.

The SPT is implemented as two bit lists each having one bit for each disk page on the volume. The table is locked into main memory. The first bit indicates whether the corresponding disk page currently stores a shadow page, and the second indicates whether the corresponding disk page is allocated to a current version page. The first list is used to return pages to the free disk page list for a volume during a stabilise operation; the second is used when a write fault occurs to determine whether the subject page has been previously shadowed since the last stabilise operation.

As part of an orderly node shut-down, the kernel retrieves all pages exported by it, and returns all pages it has imported. If a node crashes, it is unable to do this, and as a result there may be other nodes holding pages imported from the crashed node, or which have exported pages to the node. These other nodes may remain unaware of the demise of the crashed node for some time. An up/down protocol is used to assist with propagation of knowledge of crashed nodes. When a node receives a message indicating that a node is newly on-line, it checks whether it has any pages imported from or exported to the node. In the case of imported pages, these are immediately invalidated. In the case of exported pages, if any of these had been modified since the last stabilise operation, the volumes containing those pages must revert to the previous stable state. It should be noted that failure of the interconnecting medium appears to nodes as failure of remote nodes, and so appropriate time-out mechanisms must be employed to cater for temporary network disruptions. These mechanisms prevent, for instance, the aborting of a stabilise operation due to the temporary inability to retrieve exported pages.

Each volume has two root blocks which, in normal operation, point to the mapping tables for the last two stabilise operations. The integrity of a root block is guaranteed using Challis' algorithm, with an identical timestamp being written as the first and last words of the block. Only the older or the incorrect block is ever overwritten as the final stage of a stabilise operation. When a system restarts after a normal system shutdown or a system crash, the root blocks are read, and the mapping table pointed to by the most recent correct block is used to access the virtual pages stored on the volume. It should be noted that in normal operation only the most recent version of the mapping table actually exists because, as part of a successful checkpoint, the disk space allocated to the previous stable state is returned to the free disk space pool. To cater for pointers between volumes (which may be mounted on different nodes), a multi-volume stabilise mechanism is provided, implemented as a two phase commit [25, 39]. This requires that a set of dependent volumes must be stabilised at one time. For a large dependency graph with many volumes this could become quite expensive since all processes accessing those volumes must be stopped during the stabilise.

4.5 Garbage Collection

Since Monads modules are stored in address spaces which are never re-used, garbage collection of modules as a whole is never necessary. According to the principle of persistence by reachability, a module persists whilst-ever there is a pointer to it from another persistent object. Each module has a well-defined *owner*, and it is deletion of this owner's pointer to the module which results in deletion of the module [29]. On deletion of a module the disk space occupied by it is returned to the free disk space pool, and all capabilities for the module or the segments within it therefore become invalid.

Whilst garbage collection of the segments within a module has not yet been implemented, it is proposed that it will be achieved using a generation-based garbage collector [47]. This will operate on a per-module basis, and will be simplified by the fact that

(1) inter-module pointers are not permitted,

(2) segment capabilities are segregated and thus easily identified, and

(3) process stacks with pointers into a module may be easily identified using data structures maintained within the module.

The generation-based nature of the garbage collector will reduce its impact on system performance because transient data, which is typically short-lived, will be collected frequently. Garbage collection of long-term data, which is more expensive, is less productive in terms of space gained, and will thus occur much less frequently.

4.6 Protection

Conventional virtual memory systems provide each process with its own well-defined process address space in which all data available to the process is stored. Such a scheme allows each process to directly address data in the full range of addresses provided by the host machine whilst also preventing access to that data by other processes. The Monads-DSM, on the other hand, stores all data, including process stacks, in a single virtual memory space. Monads-DSM processes execute together within this single memory space, and are not restricted in terms of ability to address data by any form of per-process addressing environment. This scheme provides for much less complicated sharing of data between processes than afforded by conventional systems, but has no provision for privacy of data. Such privacy is achieved through the use of two-level *capability-based* protection [30]. Capabilities are created by the system itself, and are stored in segregated areas of the virtual memory space. As a result capabilities cannot be manufactured or arbitrarily modified by user processes.

As described in section 4.1, the first level of protection controls access to *modules*, and the second level controls access to the *segments* of these modules. Module capabilities define the module, access rights afforded by the capability in terms of available module interface routines, permissions regarding propagation of the capability itself, and advisory information about the actual storage location of the module. Segment capabilities define the offset within module of the start of the segment, the length of the segment, and access rights.

4.7 Evaluation

The Monads-DSM provides efficient multi-user access to a persistent store. The system implements the distributed server model, which reduces the possibility of bottlenecks occurring at server nodes. Distribution of the page server function provides greater resilience to single node failure. The DSM can handle movements of volumes between nodes and modules between volumes in a manner that is completely transparent to users. This transparency is achieved even though the location information embedded in virtual addresses is used to obtain virtual memory pages. As a result all module capabilities held by users of the system continue to enable access to such a moved module. Such movement may involve relocation of the entire volume on which the module is stored, or transfer of the individual module between volumes.

Enforcement of the coherency protocol is also distributed. This protocol guarantees that nodes have a coherent view of the virtual memory. The system can, however, experience page thrashing if a number of nodes concurrently attempt a series of writes to the page. Such a page would be constantly transferred between the nodes whilst not being in any of the node's main memory long enough to be used by it. Initially it appears that higher level synchronisation mechanisms would prevent such thrashing. However, because page and segment boundaries are decoupled, there is no guarantee that the nodes are attempting to write to logically related data in the page, because they may in fact be attempting to write to different segments. In such a situation higher level synchronisation mechanisms alone would not prevent page thrashing. The likelihood of page thrashing can be greatly reduced by guaranteeing a writer a small time interval (a few milliseconds) of uninterrupted write access to a page. This can be implemented by the writing node, which waits the appropriate interval before acting on a request for return of the page.

Stability of the store is achieved with a minimum of network traffic overhead, and uses existing memory coherency protocol messages. In achieving stability, the service offered to the remote user is equivalent to that offered to local users, in that, at worst, modifications made since the last checkpoint for a volume on a failed node are lost. The scheme guarantees the security of data within the system. The use of time-out periods means that temporary interruptions to the physical network media can occur without loss of data. Where cross references exist between volumes, the volumes are stabilised together utilising a two phase commit protocol over the network. When combined with an appropriate higher level transaction mechanism, it should be possible to roll forward to recover most modifications made to a volume between the last checkpoint on the volume and a system failure.

5 Conclusion

In this paper we examined two examples of the implementation of distributed persistent stores. The foundations for these implementations were fundamentally different: the Napier store is constructed above an off-the-shelf hardware and operating system platform; the Monads store utilises a custom-built architecture. A major advantage of the Napier approach is that the researchers avoided the years of development and debugging effort inherent in the construction of a new architecture, and could immediately concentrate on construction of the persistent store. In addition, the store was designed with portability in mind, and has been ported to higher performance platforms as they became available. Of importance from the research viewpoint is the fact that the store may be easily made available to other researchers because its implementation is entirely achieved in software. In contrast the Monads DSM is constructed above custom hardware which has poor performance in comparison with current workstations, and which is not widely available to other researchers. The next generation of the architecture, the Monads MM [41], is expected to provide very high performance, but is still being developed. Comparisons of raw performance, of course, should be tempered by the fact that the Monads architecture provides explicit support for the persistence and distribution paradigms, allowing a much more efficient implementation. The Monads system is also superior to conventional architectures in areas such as addressing and protection.

Different distribution paradigms are implemented by the stores: the Napier store utilises a centralised page server, whilst the Monads page server function is distributed across all nodes with attached disks. As a result of this the implementors of distributed Napier were shielded from issues such as determination of the appropriate server for a particular page of data - all pages were obtained from the central server node. The implementors of the Monads DSM, on the other hand, had access to the basic architecture and the ability to change it. This fact, coupled with the wide virtual address size provided by the architecture, allowed the Monads page server function to be implemented in a natural way across all nodes capable of storing data on disk. The wide virtual addresses supported by Monads allows modules (large objects) to be sparsely stored in it without requiring the re-use of addresses. A consequence of this is that, for the lifetime of a system, it is not necessary to garbage collect the entire store to release virtual memory addresses - garbage collection is only necessary to release disk blocks allocated to virtual pages occupied solely by unreferenced data. This garbage collection occurs within modules, and the architecture facilitates it by limiting the location and extent of pointers. In contrast, the Napier scheme provides a relatively small virtual memory space which requires constant garbage collection involving the movement of objects and thus complex pointer manipulation. Such garbage collection is, however, restricted in its area of impact by segregation of newly created objects into process-local heaps.

Another area of difference is the notion of protection of access to data implemented by the stores. The Distributed Napier system relies on the existence of trusted compilers which prevent the generation of illegal addresses. This technique has adverse implications on the use of mixed languages in the development of programs which execute within the store. Protection in the Monads system is provided through architectural support for two levels of capabilities controlling access to large and small objects respectively. These capabilities control both access to data and the mode of that access. Capabilities are stored in segregated regions of the virtual memory space, and may be created or modified only by the system kernel.

Notwithstanding these basic differences between the stores, a number of similar techniques were used in their construction. Data coherence is achieved using a multiple-reader/single-writer protocol for both implementations. The stability schemes are both based on after-look shadow paging with atomic commit achieved using Challis' algorithm. Finally, both systems implement persistence by reachability, with the name of an object being based on the virtual address at which the object is stored.

References

1. "802.3: Carrier Sense Multiple Access with Collision Detection", IEEE, New York, 1985.
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. Acceta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M. "Mach: A New Kernel Foundation for Unix Development", *Proceedings, Summer Usenix Conference*, pp. 93-112, 1986.
4. 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.
5. Atkinson, M. P., Bailey, P. J., Cockshott, W. P., Chisholm, K. J. and Morrison, R. "POMS: A Persistent Object Management System", *Software Practice and Experience*, 14(1), pp. 49-71, 1984.
6. Atkinson, M. P., Chisholm, K. J. and Cockshott, W. P. "CMS - A Chunk Management System", *Software Practice and Experience*, 13(3), pp. 259-272, 1983.
7. 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.
8. 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.

9. Brown, A. L. "Persistent Object Stores", Universities of St. Andrews and Glasgow, Persistent Programming Report 71, 1989.

10. Brown, A. L. and Cockshott, W. P. "The CPOMS Persistent Object Management System", Universities of Glasgow and St Andrews, PPRR-13, 1985.

11. Brown, A. L., Dearle, A., Morrison, R., Munro, D. and Rosenberg, J. "A Layered Persistent Architecture for Napier88", *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. 155-172, 1990.

12. Challis, M. F. "Database Consistency and Integrity in a Multi-user Environment", *Databases: Improving Useability and Responsiveness*, pp. 245-270, 1978.

13. Cheriton, D. R. "The V Distributed System", *Communications of the ACM*, 31(3), pp. 314-333, 1988.

14. Cockshott, W. P., Atkinson, M. P., Chisholm, K. J., Bailey, P. J. and Morrison, R. "POMS: A Persistent Object Management System", *Software Practice and Experience*, 14(1), 1984.

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

16. Connor, R., Brown, A., Carrick, R., Dearle, A. and Morrison, R. "The Persistent Abstract Machine", *Proceedings of the Third International Workshop on Persistent Object Systems*, ed J. Rosenberg and D. M. Koch, Springer-Verlag, pp. 353-366, 1989.

17. 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.

18. Dennis, J. B. and Van Horn, E. C. "Programming Semantics for Multiprogrammed Computations", *Communications of the A.C.M.*, 9(3), pp. 143-145, 1966.

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

20. Farmer, W. D. and Newhall, E. E. "An Experimental Distributed Switching System to Handle Bursty Computer Traffic", *Proceedings of the ACM Symposium on Probabilistic Optimisation of Data Communications Systems*, pp. 1-33, 1969.

21. Guy, M. R. "Persistent Store - Successor to Virtual Store", *Proceedings, A Workshop on Persistent Object Systems: their Design, Implementation and Use*, ed R. Carrick and R. Cooper, pp. 266-282, 1987.

22. Harland, D. M. "REKURSIV: Object-oriented Computer Architecture", Ellis-Horwood Limited, 1988.

23. Henskens, F. A. "A Capability-based Persistent Distributed Shared Memory", PhD Thesis, University of Newcastle, N.S.W., Australia, 1991.

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

25. 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.

26. Henskens, F. A., Rosenberg, J. and Keedy, J. L. "A Capability-based Distributed Shared Memory", *Proceedings of the 14th Australian Computer Science Conference*, pp. 29.1-29.12, 1991.

27. Johnston, G. M. and Campbell, R. H. "An Object Oriented Implementation of Distributed Virtual Memory", *Proceedings of the USENIX Workshop on Distributed and Multiprocessor Systems*, pp. 39-58, 1989.

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

29. Keedy, J. L. "Support for Software Engineering in the MONADS Computer Architecture", Ph.D. Thesis, Monash University, 1982.

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

31. 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.

32. Koch, B., Schunke, T., Dearle, A., Vaughan, F., Marlin, C., Fazakerley, R. and Barter, C. "Cache Coherence and Storage Management in a Persistent Object System", *Proceedings, The Fourth International Workshop on Persistent Object Systems*, pp. 99-109, 1990.

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

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

35. Needham, R. M. and Herbert, A. J. "The Cambridge Distributed Computing System", Addison Wesley, London, 1982.

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

37. Pierce, J. R. "Networks for Block Switching of Data", *Bell System Technical Journal*, 51, 1972.

38. 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.

39. 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.

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

41. 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.

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

43. Stumm, M. and Zhou, S. "Algorithms Implementing Distributed Shared Memory", *Computer*, 23(5), pp. 54-64, 1990.

44. Tam, M., Smith, J. M. and Farber, D. J. "A Taxonomy-based Comparison of Several Distributed Shared Memory Systems", *Operating Systems Review*, 24(3), pp. 40-67, 1990.

45. Thatte, S. M. "Persistent Memory: A Storage Architecture for Object Oriented Database Systems", *Proceedings of the ACM/IEEE International Workshop on Object-Oriented Database Systems*, pp. 148-159, 1986.

46. Traiger, I. L. "Virtual Memory Management for Database Systems", *Operating Systems Review*, 16(4), pp. 26-48, 1982.

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

48. 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.

49. Xerox Corporation "Courier: The Remote Procedure Call Protocol", *Xerox System Integration Standard 038112*, 1981.