keywords

persistence, distribution, cache coherence, object lifecycle, heap management, garbage collection, Napier.

Cache Coherency and Storage Management† in a Persistent Object System

Bett Koch, Tracy Schunke, Alan Dearle, Francis Vaughan,
Chris Marlin, Ruth Fazakerley & Chris Barter.

Department of Computer Science
The University of Adelaide
G.P.O. Box 498, Adelaide
South Australia 5001
Australia

{bett,tracy,al,francis,marlin,chris}@cs.adelaide.edu.au

Abstract

A distributed architecture for the support of programs written in the persistent programming language Napier is described. The architecture consists of a central server containing the stable persistent store and a collection of clients, each executing Napier processes. Since each client has a cache of objects, some of which may be shared with other clients, a protocol is required to ensure that the caches are coherent and that any access of an object will be to the most up-to-date copy. This architecture is explicated by following the lifecycle of an object from its 'birth' inside a client, through its life in the persistent store and its migration into other clients. Using this vehicle, the coherency protocol and client/server architecture are illustrated and explained.

1 Introduction

This paper describes a distributed architecture designed to support applications written in the persistent programming language Napier [1]. The Napier programmer views the world as a graph of strongly typed, stable persistent objects known as the persistent store. In this domain of discourse, all the physical properties of data have been abstracted over; examples of such properties include the location of the data, i.e. whether it is on disk or in RAM, and on which machine it resides, and how long the data exists. The Napier programmer has no control over the store — the underlying system manages the creation, movement and garbage collection of all objects.

Here we describe progress on an implementation of a persistent system based upon the well known multiple clients and single server model shown in Figure 1. In this system, a number of clients execute concurrently against a stable persistent store, managed by a single server, using a coherency protocol that guarantees data integrity. Each client corresponds to a single process executing a Napier program. These programs execute in an environment that is robust and guarantees correct execution, regardless of the failure of parts of the system.

Napier programs are compiled into Persistent Abstract Machine (PAM) code and it is an invocation of the PAM that executes the programs. One of the most notable features of the PAM [2,3] is that it is

---

† In *Implementing Persistent Object Bases: Principles and Practice* {*Proc. Fourth International Workshop on Persistent Object Systems*, Martha's Vineyard, Massachusetts, September 1990}, A. Dearle, G. M. Shaw and S. B. Zdonik (Eds.), pp.103--113 (Morgan Kaufmann Publishers, Inc., San Mateo, California, 1991).

constructed entirely upon a heap-based storage architecture. The persistent store is therefore implemented as one very large heap in which all Napier objects, including process objects, reside. Each client contains a PAM interpreter, and is provided with an area of the persistent store within which to place newly created objects; this area is termed a *local heap*.

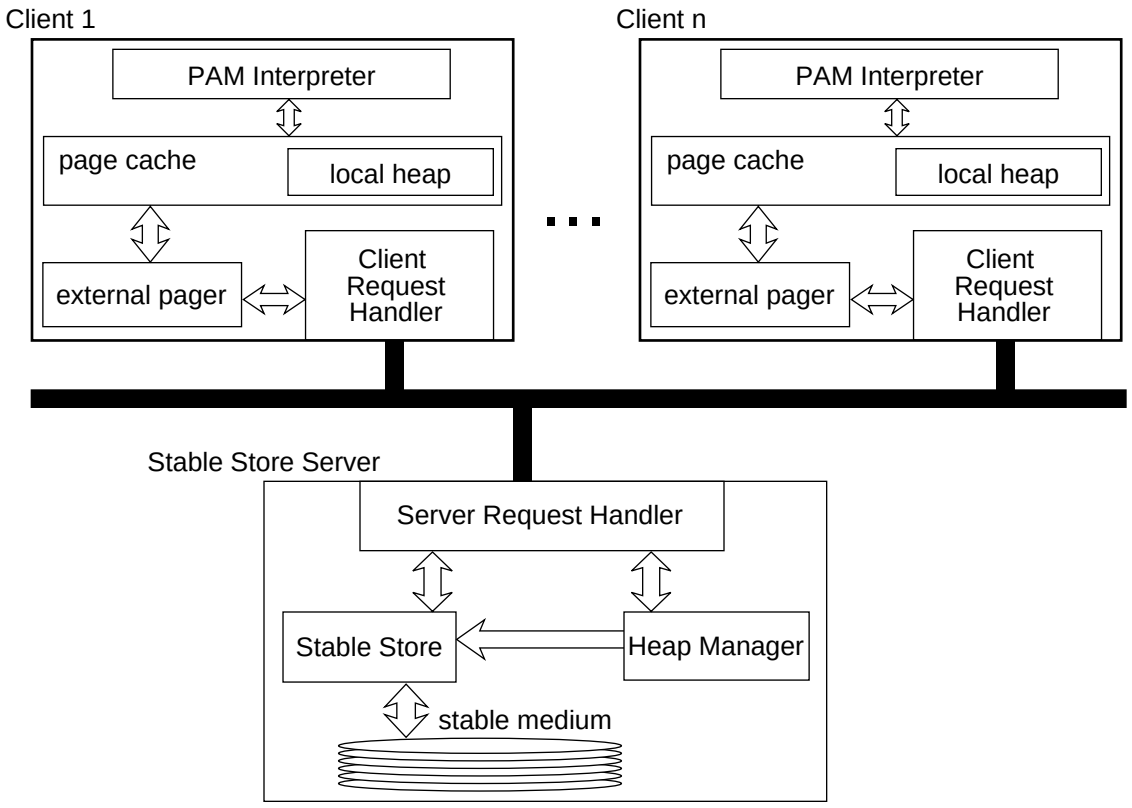

Figure 1. A persistent architecture to support distributed execution

2 The object lifecycle

In this paper, we use the metaphor of the life cycle of an object. We will follow the progress of a Napier object from its 'birth' inside a client, through its life in the persistent store and its migration into other clients. Using this metaphor, the coherency protocol and client/server architecture are illustrated and explained.

The example shown in Figure 2 manipulates two object types, namely *domicile* and *animal*. The program creates an instance of each type, makes some assignments to one of the objects, makes another assignment to some data in the persistent store, and finally makes one of the newly created objects reachable from the persistent store. The program is explained more fully in the following sections.

```
type domicile is structure( name, location : string )
type animal is structure(   name : string ;
                            home : domicile )

let smallPond = domicile(    "pond", "Adelaide" )

let afrog = animal( "tadpole", smallPond )

afrog( name ) := "frog"

use PS() with bigPond : domicile in
        afrog( home ) := bigPond

in PS() let Kermit = afrog
```

Figure 2. A Napier program

2.1. Creating local objects

Figure 3 shows a piece of code which creates an object of type *domicile* named *smallPond*, and another called *afrog* of type *animal*. As a result of running this code on Client A, five objects are placed in its local heap, corresponding to the structures denoted by *smallPond* and *afrog*, and the three strings “pond”, “Adelaide” and “tadpole”.

```
let smallPond = domicile(          "pond",
                                   "Adelaide" )
let afrog = animal( "tadpole", smallPond )
```

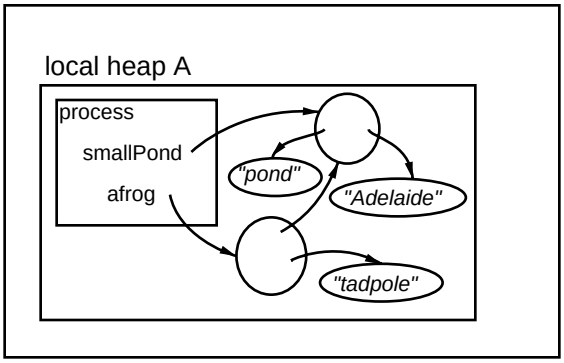

Figure 3. Creating local objects

Local objects are freely modifiable. Assignments may be made to fields of structures; for example, the *name* field of the *animal* object denoted by *afrog*, may be updated as shown in Figure 4 below. Note that this update results in the string “tadpole” being unreachable from the running process. This object will end its life in the local heap.

```
afrog( name ) := "frog"
```

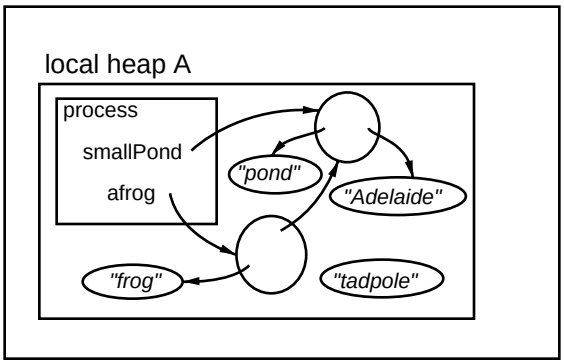

Figure 4. Modifying local objects

2.2 Accessing the persistent store

Any Napier program can access objects which exist in the persistent store by calling the predefined function *PS* to obtain the *root of persistence*. Local objects may refer to such objects. If we assume that an object called *bigPond* of type *domicile* exists in the persistent store and is reachable from the root of persistence, the migration of a frog from a small pond to a big pond may be modelled by the language-level assignment shown in Figure 5.

```
use PS() with bigPond : domicile in
  afrog( home ) := bigPond
```

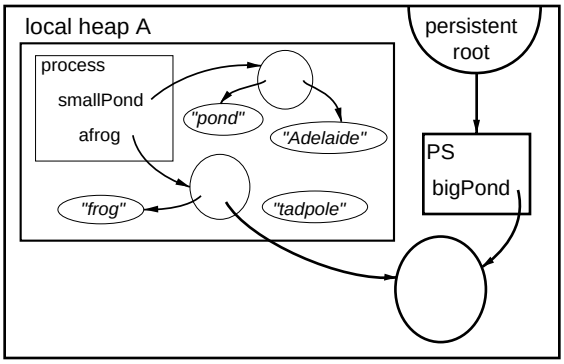

Figure 5. Object field assignments

2.3 Sharing in the persistent store

Independently, within another client, another instance of type *animal* may be created which also contains a reference to *bigPond*. Clients A and B now share references to (and possibly copies of) persistent data.

```
use PS() with bigPond : domicile in
      let bfrog = animal( "frog", bigPond )
```

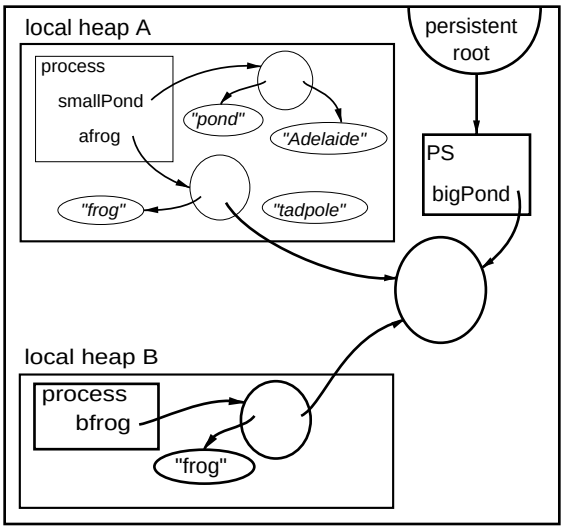

Figure 6. Shared references

2.4 Making local objects persistent

One way of making a local object available to other users of the persistent system is to make a name, and therefore a reference to that object, public. The declaration of the object *Kermit* in the root environment shown in Figure 7 achieves this.

in PS() **let** Kermit = afrog

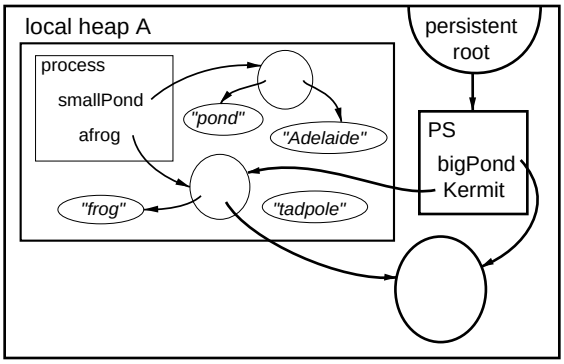

Figure 7. References from persistent to local objects

However, such an assignment causes problems as the isolation of a client's local heap from external references must be preserved. Should another client wish to access the persistent object associated with the name *Kermit*, known as *afrog* in the context of Client A, the object must be removed from its place of origin in Client A's local heap. This mechanism is shown in Figure 8.

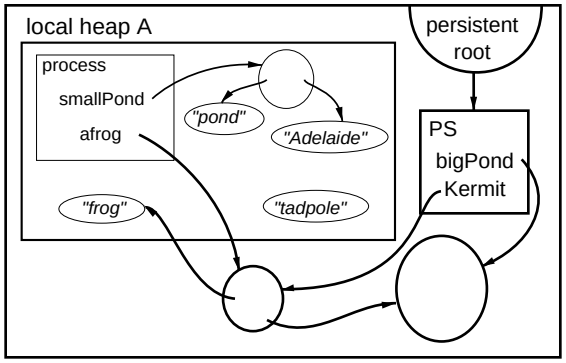

Figure 8. Object Migration

3 Characteristics of the architecture

Although the example shown in Figure 2 is obviously contrived, it is however typical of many Napier programs. Firstly, notice that the program constructs a graph, in this case a trivial one; once the graph is constructed, it is linked into the persistent object graph. This kind of behaviour is common in programs written in persistent programming languages. In fact, linking a newly constructed data structure into an existing persistent structure is often the last act of a program before ceasing execution.

If this is typical behaviour for persistent programs, it would suggest that making assignments from persistent data to newly created data is relatively infrequent. Furthermore, it has been shown by Loboz [4] that most objects in persistent object stores are not modified. If the persistent address space is large, we believe it is also safe to assume that most data will not be shared. The strategies described in Section 7 depend on these assumptions.

We have attempted to design a system in which there is only one level of object addressing like that used by Thatte [5]. This contrasts with other persistent systems which use the now well known technique of persistent identifier (PID) translation [6,7]. The benefit gained is the opportunity to exploit address translation hardware.

The strategies used attempt to optimise what we expect to be common access patterns. Whether or not these strategies are advantageous will only be revealed through measurement once the system is complete. The system described in the following sections takes advantage of paging hardware, utilises the external paging facilities provided by the Mach operating system [8] and avoids the use of indirection tables and PID address translation (pointer swizzling) implemented in software.

4 Creating local objects

The object space in this system is shared by all clients and comprises a single 4 gigabyte address space known as the *persistent address space*. Objects reachable from the persistent root are termed persistent, and the pages used to hold these objects are termed *persistent pages*. The persistent address space is maintained by the *Stable Store Server*, shown in Figure 1, and it is within this address space that instances of PAMs execute.

Each client views the same address space of objects within which references may be made by directly addressing the objects. At any time, each client may contain part of the total persistent address space in a *page cache*.

The local heaps described in Section 1 reside inside page caches. Each local heap corresponds to a Napier process and is constructed from a contiguous set of pages from the persistent address space. Local heaps are small enough to always remain resident within the client's page cache during normal execution. Experience with earlier implementations has shown that significant improvements in performance may be obtained if the local heap area is rarely paged. Furthermore, greater locality of reference may be obtained in relation to larger heaps that do not use the local heap model. This can result in improved performance from better page fault behaviour and improved processor cache utilisation.

All objects are created in a local heap. In Figure 3, the object representing the string “tadpole” will be unreachable long before it gains any rights to long-term persistence. In fact, many Napier objects are short lived; this is exacerbated by the fact that Napier is a block retention language [9] and that activation records are allocated by the PAM from the heap rather than from a stack.

If transient objects are confined to a localised area, they may be distinguished from persistent objects resident in the total persistent address space and consequently garbage collected locally at low cost. Local heaps may be safely garbage collected provided that no external references (from other processes or the server) point into them. Fortunately, the creation and export of such pointers are easily detected, making this technique tractable. This is discussed further in Section 7.

The implementation permits, and requires, a single external pointer into local heaps. This pointer is to the *process header* object representing the process executing in the local heap and is used as a root for local garbage collection. Process headers are reachable from the root of persistence making processes restartable in the event of a failure.

5 Accessing the persistent store

The distinguished object known as the root of persistence is stored at a known address and is accessible by any process. A typical Napier program makes use of many objects found in the persistent store; examples of such objects include a browser [10], a window manager [11] and the Napier compiler. A process may traverse the transitive closure of objects reachable from the persistent root in order to access these tools; in so doing, it constructs references to objects that are outside its local heap.

An example of this mechanism is shown in Figure 2, in which the object denoted by *bigPond* is reachable from the root environment. The code fragment in Figure 5 results in the creation of a pointer from the locally created object, denoted by *afrog*, to a member of the persistent heap (*bigPond*). Recall that the objects named *afrog* and *bigPond* reside in the same persistent address space and therefore the reference to *bigPond* from *afrog* is a direct pointer stored within the object.

When an attempt is made to access an object not resident in the page cache, a page fault is generated. Under Mach, the user is permitted to provide a process called an external pager which services page faults. If an external pager is associated with a user process, the Mach kernel will forward page fault exceptions to that external pager, which will return the required data (in the case of a read fault) or may write the data to some stable medium (for a write fault).

The external pager, upon receipt of the page fault, communicates with the Stable Store Server via the *Client Request Handler*. The Client Request Handler handles all communications with other clients and the

Stable Store Server. Once the appropriate page is delivered and placed in the client's address space, the access may proceed normally.

6 Sharing in the persistent store

The coherency algorithm addresses the problem illustrated in Figure 6, where persistent page copies are required by a number of clients. Further problems arise when one of those clients attempts to modify such a page. Two finite state automata have been developed to implement a coherency protocol between the clients and the server. These are based on the following principles:

- a page copy may be held by more than one client for read-only access, and
- only one client may hold write permission for a particular page.

To enforce these access restrictions, the Stable Store Server referees all page requests from connected clients. It determines whether pages are available for read or write access and distributes the requested pages accordingly. The server, therefore, maintains the majority of data structures used to ensure coherency. For each page, a data structure records those clients which hold up-to-date copies of that page; this list is known as the V-list (valid list).

If the Stable Store Server receives a request from a client to read a persistent page, it must determine if it is capable of directly supplying a copy of that page. If it can, it immediately sends the page to the requesting client; this is illustrated in Figure 5 where a previously unrequested page is requested by Client A. If the page may have been modified with respect to the server's copy, it must forward the read request to a client with a valid copy of that page. Such a client may be identified by examining the V-list associated with that page.

In Figure 6, the server is unable to supply the page copy as the page is held by a single client which may freely modify the copy of the page in its local page cache without arbitrating with the server for write permission. The server must therefore forward client B's read request to Client A which will, in turn, forward the potentially updated copy of the page to client B.

Note that if a client is known to hold the only copy of a page, it may update that page without requesting permission from the server. However, if the page is unmodified, the client must inform the Stable Store Server of its intention to modify the page. This mechanism will represent a considerable performance increase if the assumptions made in Section 3 are correct.

Copies of persistent pages in clients' page caches are initially write protected. This causes an access exception to be raised if a PAM attempts to modify a shared page. Whenever this occurs the Client Request Handler requests the Stable Store Server for write permission for that page. The server must ensure that all valid page copies are removed from other clients before it grants write permission to the requesting client. The server sends invalidation signals to all V-list members except the client which requested modification permission for the page. Upon receipt of all the invalidation acknowledgements from those clients requested to invalidate, the Stable Store Server informs the original requesting client that write permission is granted.

7 Making local objects persistent

Continuing the progression through the events an object may experience in its lifetime, we now consider the situation where a persistent object, resident in a persistent page cached within a client, points to an occupant of that client's local heap. As illustrated in Figure 7, this arises at the language level when an assignment is made from an object which is already persistent, to data created locally. At the implementation level, this assignment means that data created by the interpreter within the confines of its local heap is now reachable from an external point, other than its process header. For the reasons stated earlier, chief among which is preserving the ability to garbage collect the local heap area independently, it is important to:

- be able to detect the construction of pointers from persistent pages to locally created objects, and
- devise a mechanism whereby no other client will obtain a page while it contains references into local heap areas.

Using a scheme derived from generation-based garbage collection [12], another set of previously unused persistent pages, termed *copy-out pages*, is maintained by a client. In addition, the set of persistent pages which reference local objects is recorded; following Ungar, this set is termed the *remembered set*. New members may be added to this set when an assignment occurs at the machine level. For example, the statement

in PS() **let** Kermit = afrog

results in the addition of the address, representing *Kermit*, to the remembered set.

If a local heap always comprises *contiguous* pages, the interpreter can make a simple check at such a time: if the source address lies outside the range reserved for the local heap, and the assignment target lies within it, the source and its corresponding page identifier are placed in the remembered set. In Figure 9, pages P1 and P2 contain objects which reference the local objects A and D. These represent the instances of animal and domicile structures from Figures 3 to 8. The remembered set records the addresses of the pointers representing *Kermit*, *Muppet* and *Rymill*. The last two references could be the result of the following implementation level assignments:

in PS() **let** Muppet = afrog
in PS() **let** Rymill = smallPond

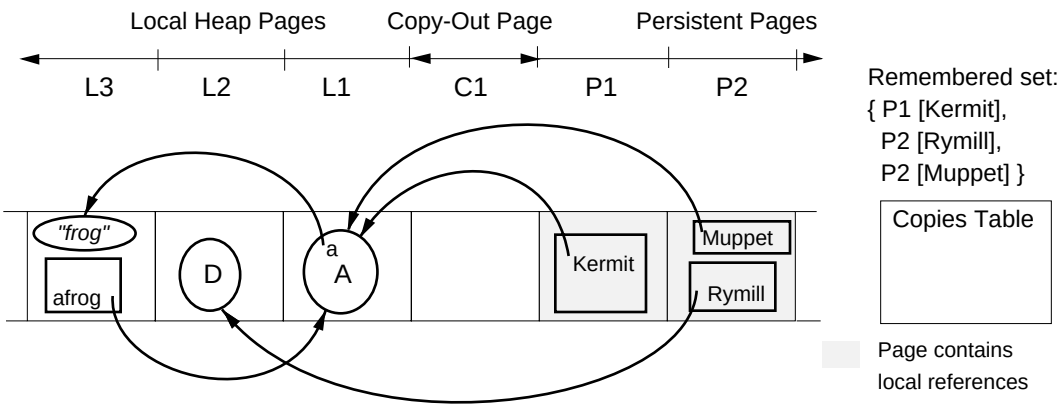

Figure 9. A client's page cache

The copy-out pages are used when an external request arrives at the client for a page belonging to the remembered set. The basic strategy is to copy the local heap object (or objects), referred to by the requested page, into a copy-out page. All references to the objects must eventually be changed to reflect their new locations within the persistent virtual address space. References from other persistent pages may be found efficiently, and updated immediately, by searching the remembered set. However, references from local heap pages may only be detected by traversing the local heap.

Such a traversal is expensive in terms of time, delaying the required page supply; it also necessitates halting the interpreter, which is potentially running concurrently at this time. Instead, this operation is postponed, and another data structure called the *Copies Table* is utilised. This table holds (*old*, *new*) address pairs for objects which have been duplicated. Once moved, the object itself must be scanned for local addresses among its pointer fields; should any be found, appropriate entries are inserted in the remembered set. The old and new addresses of the object are placed in the Copies Table. The page may now be safely exported since it no longer contains references to the client's local heap.

Figure 10 shows the situation where page P1 has been prepared for export by the above procedure. Object A has been moved into the copy-out page C1, and the two persistent locations which contained references to A, now point to the object A'. Finally, the mapping between A and A' has been entered in the Copies Table.

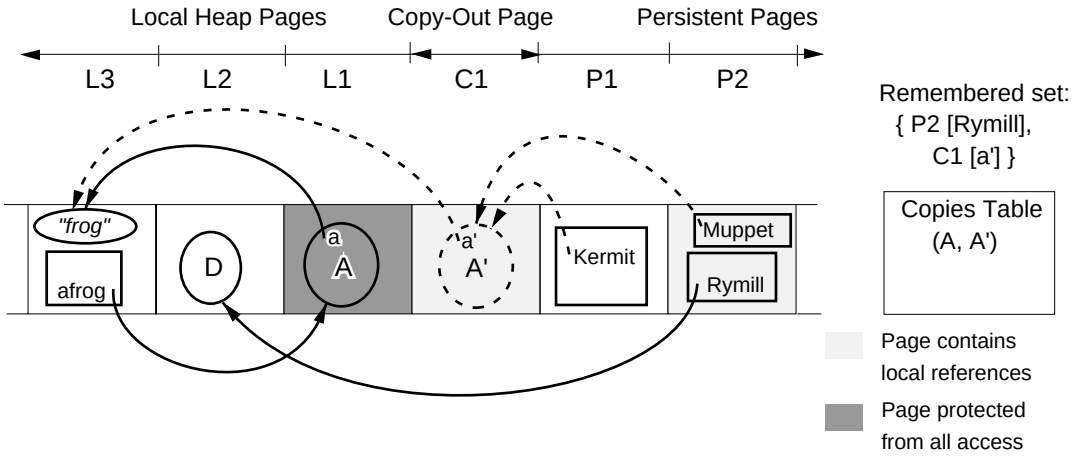

Figure 10. Copying out a local object

The original copy of a duplicated object cannot be classed as garbage until those references to it from within the local heap are altered. To trigger the alteration procedure, the local heap page containing the original copy is completely protected from all accesses. Page L1 in Figure 10 is depicted as protected since it contains the old copy of object A.

If the interpreter attempts to access such a page, the external pager will be invoked with an access exception. Since the interpreter has halted, the local heap can now be fully traversed to find and update all references to copied objects; the relevant entries in the Copies Table can subsequently be deleted. If the interpreter does not attempt to access this page before the next local garbage collection, all references from within the local heap to relocated objects are updated at this time, and the Copies Table becomes empty.

In Figure 11, the pointer from *afrog* has been updated, the mapping (A,A') has been removed, and the protection on page L1 has been lifted. In this way, objects migrate from initial creation as part of a process' local heap, to general availability as part of the shared persistent store.

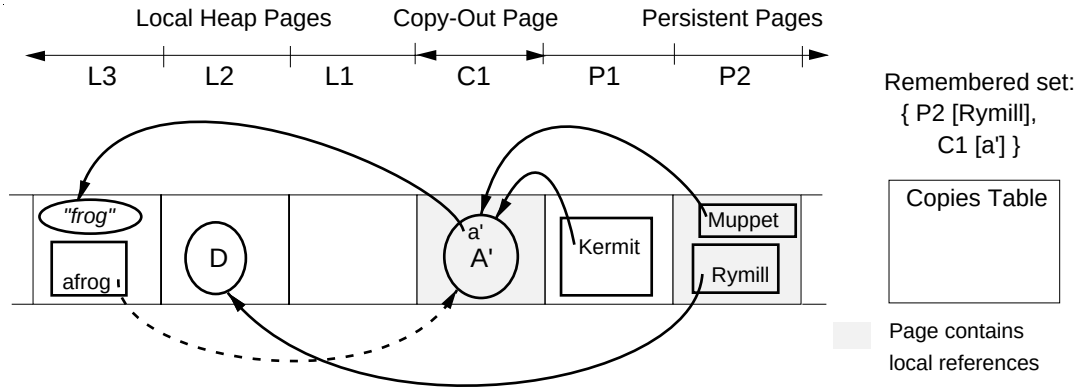

Figure 11. Completing reference updates

Different clustering strategies may be modelled by altering the way in which the copy out process is implemented. Suppose the copy-out page C1 in Figure 11 is required by another client; the procedure described above would be repeated. If object A formed part of a linked data structure, it is highly probable that C1's data would soon be of non-local interest. It may be advantageous, when initially required to copy an object, to immediately copy its transitive closure as well. Such a strategy caters for a depth-first pattern of access to objects.

Alternatively, it may be better to anticipate future export requests for all pages in the remembered set, and, when one such page is requested, move *all* objects directly referenced from this set. Adopting the first option would cause both the structure A and the string object "frog" to be moved if page P1 was requested; adopting the second would result in the relocation of both the structures A and D in a breadth-first manner. Experimentation and further measurements regarding typical object access will be needed to determine an optimal strategy.

8 Stabilisation

We have described a distributed object space managed by a central server. The central server provides a stable store in which a self consistent version of the object space is maintained. To ensure stability may always be achieved, rollback is made possible by writing data to disk using a shadow paging technique [13]. In general terms, this scheme involves maintaining original page versions along with the new. As the final stage in a stabilisation, a single disk-page write updates a mapping, reflecting which version is to be treated

as the original henceforth [14]. Until this point, the previous stable state is completely preserved, and indicated by the old mapping.

The Stable Store Server maintains those structures needed to correctly roll back the execution state of interdependent clients should failure occur in any part of the system. Those parts of the system that can continue without jeopardising the store's integrity are unaffected.

A client which holds modified pages may pass copies on to another client as a result of read requests. Thus, clients may become dependent on one another by virtue of having seen the same data, which has been modified with respect to the Stable Store. We term such dependent clients *associates* and a set of mutually dependent clients an *association* [15].

It is important to note that associations are dynamic in nature, with clients being added and associations merging over time. Associations, which are maintained by the Stable Store Server, contain a list of page identifiers, which identifies those pages modified by members of the association since the last stabilise.

Any client may initiate a stabilise operation. At such a time, the server notifies the instigating client's association to stabilise in order to bring the stable store into a new consistent state. Each of the associates must send back copies of all modified pages held in their local cache. It is not necessary to write back pages until this time. Due to the nature of associations (that is, a client may belong to only one association and a modified page belongs to only one association's page list), several independent stabilisation cycles may be in progress at any time.

9 Conclusions

A distributed architecture designed to support applications written in the persistent programming language Napier has been described. Objects are manipulated within caches inside clients, each of which are serviced by a central server; the architecture supports the sharing of objects between clients. Thus, the question of cache coherency arises and a sophisticated cache coherency protocol has been designed for the system.

Alternative schemes for sharing virtual memory between distributed clients have been proposed [16,17]. In [18], the underlying data repository moves between *stable states* through *checkpoint* operations. A checkpoint involves flushing those pages onto disk which have been modified since the last checkpoint. A new stable state is achieved when a known set of such pages has been thus secured, and any record of their original contents, as held in the previous stable state, can be safely disposed. Wu and Fuchs [19] describe a system whereby checkpoints are carried out on individual nodes (i.e. clients), as soon as another node requests the use of any updated data. A major concern of this work has been to limit rollback propagation, so that the failure of any client only affects that client.

The idea of having local heaps originates in architectures such as POMS [6] and CPOMS [7] that perform PID translation. These systems have the advantage of being able to perform local garbage collection, but pay a heavy cost penalty in the form of software address translation.

Our approach to distributed support for persistent objects has been illustrated through the description of the lifecycle of an object from its ‘birth’ inside a client, through its life in the persistent store and its migration into other clients. Our scheme also shows potential for clustering objects in a page based persistent address space, with all the attendant benefits. The architecture exploits hardware address translation through the external pager mechanism provided by the Mach operating system. This scheme capitalises on the advantages of a local heap. In addition to potentially improving performance through locality of reference, the likelihood of having to perform garbage collection over a large persistent address space is minimised.

Acknowledgements

This work was partially supported by Australian Research Council grant number 4900-6830-1000. We would like to thank the Persistence Project at the University of St Andrews for their continuing cooperation in this work; in particular, we acknowledge the contributions of Fred Brown to whom this work owes a great deal.

We would also like to thank the National Parks and Wildlife Service of South Australia, particularly the staff of the Morialta Falls Conservation Park who erected a billboard showing the life cycle of a frog.

References

[1] Morrison R., Brown A.L., Connor R. & Dearle A. “The Napier88 Reference Manual”. Universities of Glasgow & St Andrews Persistent Programming Research Report 77-89 (1989).

[2] Morrison R., Brown A.L., Carrick R., Connor R. & Dearle A. “The Persistent Abstract Machine”. Universities of Glasgow & St Andrews Persistent Programming Research Report 59-88 (1988).

[3] Connor R., Brown A.L., Carrick R., Dearle A. & Morrison R. “The Persistent Abstract Machine”. In “Persistent Object Systems”. Workshops in Computing, Rosenberg J. & Koch D. (eds.) pp. 279-288 (Springer-Verlag, 1990).

[4] Loboz C.Z. “Monitoring Execution of PS-algol Programs”. In “Persistent Object Systems”. Workshops in Computing, Rosenberg J. & Koch D. (eds.) pp. 353-366 (Springer-Verlag, 1990).

[5] Thatte S.M. “Persistent Memory: A Storage Architecture for Object Oriented Database Systems”. Proceedings ACM/IEEE 1986 International Workshop on Object Oriented Database Systems, Pacific Grove, California, pp. 148-159 (September 1986).

[6] Cockshott W.P., Atkinson M.P., Chisholm K.J., Bailey P.J. & Morrison R. “POMS: a Persistent Object Management System”. Software Practice and Experience, 14,1 (January 1984).

[7] Brown A.L. & Cockshott W.P. “The CPOMS Persistent Object Management System”. Universities of Glasgow & St Andrews Persistent Programming Research Report 13-85 (1985).

[8] Acceta M., Baron R., Bolosky W., Golub D., Rashid R., Tevanian A. & Young M. “Mach: A New Kernel Foundation for UNIX Development”. USENIX, pp. 93-112 (July 1986).

[9] Berry D.M. “Block Structure: Retention or Deletion?”. Conference Record of the Third Annual ACM Symposium on the Theory of Computing, Shakes Heights, Ohio, pp. 86-100 (May 1971).

[10] Kirby G. & Dearle A. “An Adaptive Browser for Napier88”. University of St Andrews Research Report CS/90/16 (1990).

[11] Cutts Q., Dearle A., Kirby G. & Marlin C. “WIN: A Persistent Window Management System”. Universities of Glasgow & St Andrews Persistent Programming Research Report 73-89 (1989).

[12] Ungar D. “Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm”. ACM SIGPLAN Notices, 9,5 pp. 157-167 (May 1984).

[13] Lorie A.L. “Physical Integrity in a Large Segmented Database”. ACM Transactions on Database Systems, 2,1 pp. 91-104. (1977).

[14] Challis M.L. “Database Consistency and Integrity in a Multi-user Environment”. In “Databases: Improving Usability and Responsiveness”. B Scheiderman (ed.), pp. 245-270 (Academic Press, 1978).

[15] Vaughan F., Schunke T., Koch B., Dearle A., Marlin C. & Barter C. “A Persistent Distributed Architecture Supported by the Mach Operating System”. Proceedings USENIX Workshop on the Mach Operating System, *to appear* (October 1990).

[16] Li K. & Hudak P. "Memory Coherence in Shared Virtual Memory Systems". ACM Transactions on Computer Systems, 17,4 pp. 321-359 (November 1989).

[17] Rosenberg J., Henskens F., Brown F., Morrison R. & Munro D. "Stability in a Persistent Store Based on a Large Virtual Memory". University of St Andrews Research Report CS/90/6 (1990).

[18] Morrison R., Brown A.L., Connor R. & Dearle A. “Napier88 Release1.1”. St Andrews University, St Andrews, Scotland (1989).

[19] Wu K.L. & Fuchs W.K. "Recoverable Distributed Shared Virtual Memory". IEEE Transactions on Computers, 39,4 pp. 460-469 (April 1990).