Casper : a Cached Architecture Supporting Persistence

Francis Vaughan, Tracy Lo Basso, Alan Dearle,
Chris Marlin, Chris Barter

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

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

Abstract

Persistent object systems greatly simplify programming tasks since they hide the traditional distinction between short-term and long-term storage from the applications programmer. As a result, the programmer can operate at a level of abstraction in which short-term and long-term data are treated uniformly. It is important that such a persistent system be capable of being used concurrently; such concurrent usage may arise because of parallel process facilities in the programming language concerned, or because of multiple users of the same persistent store. Concurrent access has not been satisfactorily supported in existing persistent store implementations and a number of significant research issues remain to be investigated. This paper describes an architecture that supports concurrent access to a shared persistent object store. The persistent distributed architecture represented by our system exploits a number of the facilities provided by the Mach distributed operating system

1. Introduction

Persistent programming is a relatively new paradigm that makes data intensive application programming significantly easier. The idea behind *persistence* [Atkinson, et al. 1983] is a simple one: data in a system should be able to persist (survive) for as long as that data is required. *Orthogonal persistence* means that all data may be persistent and that data may be manipulated in a uniform manner regardless of the length of time it persists. In other words, the right for data to survive for a long (or short) time is independent of the type of data. Programs manipulating data do so in a uniform manner, whether the data is short or long lived.

Conventional programming languages require the programmer to explicitly manage persistent data: to save data, the programmer must traverse the in-memory data structures and create an isomorphic structure in the file system. Similarly when that data is later required, the reverse process must be executed: the data in the file system must be traversed and the in memory structure re-created. Furthermore, to protect the integrity of the data, great care in programming is needed; for example, the modification of a data structure in the file system must avoid corruption even in the event of power failures or system crashes.

Using a traditional database avoids many of these problems, for example, updates to persistent data is controlled by carefully designed transaction mechanisms. However, there is a hidden cost with database systems: the representation of data in the data base seldom matches the type system used by the programming language. Therefore the programmer must still manage the conversion of data from one form into another. This problem is sometimes called the *impedance mismatch problem* [Bancilhon and Maier 1989].

Persistent programming eliminates impedance mismatch by providing a high level language in which data of arbitrary longevity (short or long) may be created, stored and manipulated. Persistent systems support long lived data objects of arbitrary complexity – such data objects may not only outlive instantiations of the program that created them, but also outlive versions of the program, or even the useful life of the program in all its versions.

This paper describes an architecture designed to support applications written in persistent programming languages. The architecture maintains an orthogonally persistent programming environment that is coherent across a number of clients and is supported by a resilient object store. It is capable of supporting the majority of persistent, algorithmic, object-oriented and applicative programming languages currently in use, as well as being sufficiently flexible to be used as an experimental platform. However, the original motivation for constructing the architecture was to support the persistent language Napier88 [Morrison, et al. 1989].

The model of persistent store supported by Casper is a conceptually infinite, shared and resilient data store. The aim of the architecture described in this paper is to support concurrent access to the persistent store by users on workstations connected by a local area network. This model of persistence has lead us naturally to a design which provides a persistent store abstraction similar to Distributed Shared Memory [Philipson, et al. 1983] with all clients sharing the

same address space (the persistent store). The needs of persistent programming make particular demands upon the maintenance of this address space beyond the conventional needs of coherency in DSM. For example, the ability to maintain a self consistent and recoverable state in the stable store is an attribute normally associated with databases but one central to the success of our design.

1.1. Napier88

One difference between Napier88 stores and traditional database systems is that in addition to the usual passive data the stable store contains meta data such as procedures and functions [Atkinson and Morrison 1985]. This allows Napier88 stores to be used as repositories for programs. It also allows data to be encapsulated within the closure of functions and procedures providing a level of abstraction not available in relational systems. Such functionality permits the persistent programming environment to subsume the roles of traditional file and database management systems.

The state of all processes executing within the store are also part of the persistent environment. Thus should some component fail due to a hardware malfunction or power failure the user processes executing in the system will continue execution upon restart.

Another difference between Napier88 systems and database management systems is that database management systems require a sequence of update operations by a user process to be contained within an atomic transaction. That is, either all modifications are completed, or none are made. Traditionally, such atomicity is achieved by either locking portions of the database, or duplicating portions, so that while a transaction is in progress, no other user process may view modified data [Eswaran, et al. 1976]. Casper does not force all data accesses to be serialised; instead, the grain of atomicity is that of the operations in the Napier88 language.

The Napier88 language was developed by the PISA project [Atkinson, et al. 1986] as a test-bed for experiments in type systems, programming environments, concurrency, bulk data objects and persistence. The Napier88 type system is polymorphic and evolved at the same time as Cardelli and Wegner [Cardelli and Wegner 1985] published their work. Many of the ideas are related to theirs and some have been borrowed from them. The philosophy is that types are sets of values from the value space. The possibility of static type checking is retained wherever possible. However, dynamic projection out of the types *any* and *environment* [Dearle 1989] permits the dynamic binding required

for true orthogonal persistence. Napier88 is unusual in that, like its predecessor PS-algol, it is a store-based language with higher order procedures and block retention [Berry 1971]. The Napier88 system consists of the language and its persistent store. This persistent store is populated with objects, some of which are used to support itself. Examples of such tools include an object browser, a window manager and the Napier88 compiler, which may be called dynamically to provide ad-hoc polymorphism and reflection.

1.2. Mach

We have implemented Casper using the Mach operating system [Acceta, et al. 1986]; Mach provides a suitable base for the system's implementation through its support for programmable page fault handling, inter-process communication, exception handling and multiple threads.

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 pages removed from the client's physical memory). This external pager mechanism implements most of the functionality needed to support the coherent persistent address space described later.

The inter-process communication (IPC) structure available in Mach permits a transparent interface to be built, independent of the physical location of the communicating parties.

Mach supports more than one thread of execution in a single virtual address space, which is exploited by the architecture described in this paper. We have found this to be especially useful in building asynchronous communication protocols, such as our cache coherency protocol.

2. Implementation structure

2.1. Overview

The architecture of our persistent system is depicted in Figure 1. A number of clients execute against a shared stable store using a coherency protocol that guarantees data integrity; client code executes in an environment that is robust and guarantees correct execution regardless of the failure of parts of the system. Each client has an interface to the *Stable Store Server*, which gives access to the stable persistent store; the interface at the client is called the *Client Request Handler*. In addition, each client contains a thread executing the users code

(compiled Napier88 programs) and a page cache which holds copies of pages required by the user program. Within a client, coherency is maintained by the Client Request Handler and the external pager.

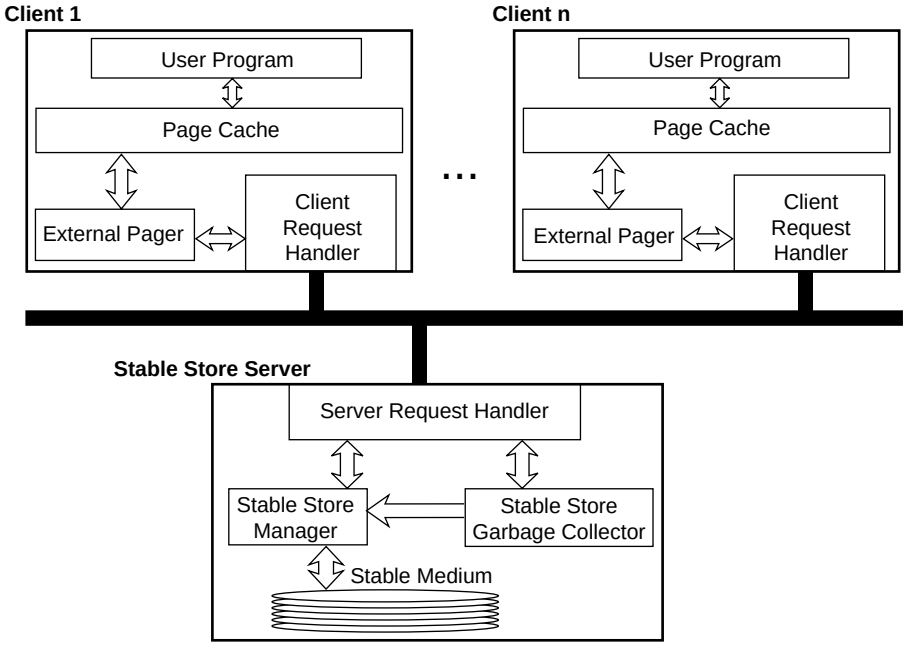

Figure 1. A distributed persistent architecture.

A *stable store* is defined by Lampson [Lampson 1981] to be a set of objects which move from one consistent state to another atomically. In the Casper system, the stable store is provided by the Stable Store Server. The Stable Store Server consists of four components: the *Server Request Handler*, the *Stable Store Manager*, the *Stable Store Garbage Collector* and the *stable medium*. At the lowest level of abstraction, the stable store is implemented using a stable medium such as disk storage. The objects resident on the stable medium are managed by the Stable Store Garbage Collector, whilst the physical pages are managed by the Stable Store Manager. Finally, the interface to the outside world is provided by the Server Request Handler.

In addition to the usual passive data found in traditional database and file systems, the Stable Store contains active data including the state of all processes executing within it. This provides the potential for restarting processes found in the persistent store should some element of the system fail. The protocol definition includes the maintenance of 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.

2.2. Stability and Coherency

As stated earlier, our system does not force accesses to the database to be serialisable. Instead, anarchic access to the store is permitted and synchronisation is provided by language level mechanisms. However, the store itself must be kept consistent, which presents two problems:

- the contents of the store may never represent an inconsistent state. Whenever a snapshot of the executing environment is made it must be done in such a way as to create a new self consistent state, or fail in the attempt and roll back the environment to the previous self consistent state.
- no process must view or act upon out-of-date data, or be able to modify a data item concurrently with another process.

In this system, store stability deals with the first aspect, and the cache coherency protocol with the latter. Store stability is the subject of Section 3, coherency is discussed in Section 4.

3. Store Stability

As described in the previous section, all access to the persistent store is controlled by the Stable Store Server. It has three main functions: managing the supply of pages upon demand to clients, ensuring that coherent versions of the pages are supplied, and, maintaining the integrity of the Stable Store. It also allocates ranges of the persistent address space to processes and garbage collects the main heap.

Since the persistent store is used as the repository for all objects shared by clients, it is imperative that the contents of the store remain stable (i.e., have the ability to survive failures). This requires the use of a reliable mechanism to maintain consistency within the stable store.

In the Napier88 system, the underlying data repository moves between *stable states* through *stabilise* operations. A stabilisation is usually initiated by user code via a predefined Napier88 function which is always guaranteed to succeed. As described in Section 3.1, stabilisation may occur asynchronously with respect to a client due to the sharing of data with other clients combined with a stabilisation request made by one of those clients.

In Casper stabilisation involves flushing those pages onto disk which have been modified since the last stabilisation. A new stable state is achieved when a

known set of such pages has been secured; any record of their contents in the previous stable state can then be safely discarded. In this way, there is always a recent, reliable state to which it is possible to return following failure in some part of the system. This is discussed in more detail in Section 3.3.

3.1. Associations

In Casper, clients may share modified data. This presents the problem of maintaining a consistent view of the persistent environment. Consider the case where two clients, α and β, both share data. If client α stabilises its state independently of β an inconsistent view of data will be created. This can be seen by considering the consequences of a system crash immediately after the stabilisation of α. α and β revert to the state at the time of their last stabilisation, and these times are different. Therefore the state of α would be consistent but β would view non shared data at the time of its last stabilisation but shared data at the later time of the last stabilisation of α.

There are three solutions to this problem:

1. prevent clients from sharing data modified with respect to the stable store,
2. make all clients stabilise together, and,
3. make interdependent clients stabilise together.

Wu and Fuchs [Wu and Fuchs 1990] describe a hardware implementation in which clients are prevented from sharing modified data. This is achieved by forcing clients to perform checkpoint operations as soon as another client requests the use of any updated data. A major concern of their work has been to limit roll-back propagation, so that the failure of any client affects only that client.

The approach of making all clients stabilise together has been implemented in Monads [Henskens, et al. 1991] , a persistent system that allows clients to share modified pages.

We adopt the solution whereby clients may share modified data and only interdependent subsets of clients must stabilise together. We term interdependent clients *associates* and a set of mutually dependent clients an *association*. It is important to note that associations are dynamic in nature, with clients being added and associations merging over time. These associations are maintained by the Stable Store Server. The maintenance of associations is described in Section 4.2.

3.2. Failure

Failures are characterised as the non-recoverable loss of either a client or the server. Such failures may be caused by power failures, operating system errors or the crash of the system itself. Such failures must be reliably detected and the system state recovered in a manner that does not compromise the integrity of the store.

Communication between elements of Casper uses the IPC mechanisms provided by Mach. Internally Mach IPC implements a virtual circuit and is able to detect the loss of any element of the system. Depending upon which element fails, different action is taken.

If a single client that is not a member of an association fails it is only necessary for the Stable Store Server to remove its record of data pertaining to that client. If a client that is associated with other clients fails, the failed client is treated similarly to the single client case, but in addition all of the other members of the association are forced to revert to the state represented in the stable store. This is their state at the time of their last stabilisation. Reversion is performed by invalidating all of the modified pages within the clients and then recovering the internal register state of the executing processes from the stable store. Reverted processes effectively travel back in time and then continue to execute, albeit without their failed associate.

Failure of the store server currently results in all of the clients terminating and requires human intervention to restart. Runnable processes are resumed from their saved state in a manner similar to that described above.

3.3. Shadow Paging Scheme

Stabilisation requires that a set of objects move from one state to another atomically. In Casper this is achieved using a *shadow paging* technique [Lorie 1977] combined with an atomic commit operation.

Shadow paging means that a dirty page may never overwrite a clean version of a page in the stable store. Instead, dirty pages are written to another location, with these copies of the pages being known as *shadow copies*. Thus, when a page is modified for the first time, a shadow location for it is allocated in the stable store. At most one shadow copy of each page exists and once a shadow copy of a page exists, that page is used for future operations. Stabilisation changes the status of shadow pages so that they are now considered the clean version and further modification of the page results in the allocation of a new shadow page. The old clean page is free to be reused. Maintaining a shadow copy permits the system

to roll back should a failure occur, since the original pages are never overwritten until after the successful completion of a stabilisation operation.

To implement such a system, a mapping must be maintained that maps the address of a persistent page to its location in stable memory (i.e., disk). This mapping table is called the *Logical to Physical map* (L-P map). As shown in Figure 2, each entry in the L-P map contains three fields: the physical page location of the stable version of the page, the location of the shadow copy of the page (if one exists) and a single bit selecting which entry holds the address of the stable page. Since the L-P map must be robust, it is natural to place it within the persistent store which it manages. Consequently, there may be two versions of the data structure – a stable version and a shadow version.

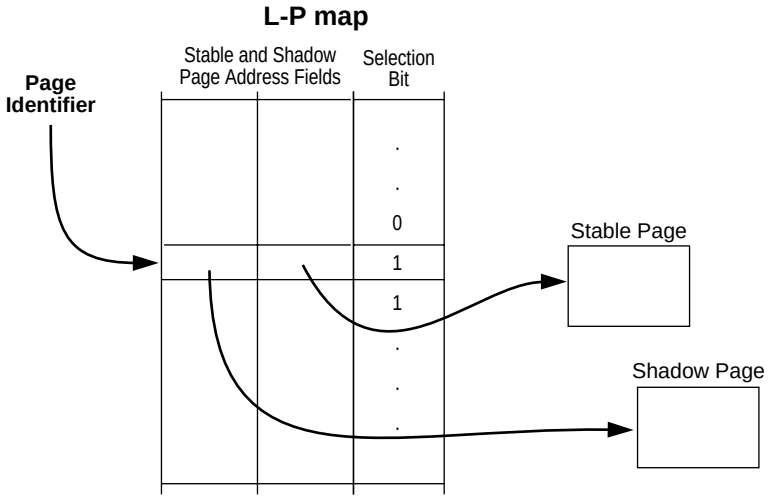

Figure 2. The L-P map.

Stabilisation requires that a new consistent stable state be created from a set of pages consisting of some newly created shadow pages and some existing stable pages. Furthermore, it must be performed in such a way that it is always possible to recover the state before stabilisation, even if a failure occurs during stabilisation.

The stabilisation sequence is as follows: the modified pages are first written from the clients to their shadow locations on disk. In the normal course of events, modified pages may also have been delivered to the stable store by the coherency mechanism if there was insufficient space for them within a client's physical memory. These pages are also written to their shadow locations and are regarded as having been written back as part of this stabilisation. Once all of an associations pages have been secured in the stable store, the selection bit on each of these pages is flipped to indicate that the shadow version is to be used as the current version once the stabilisation is complete. Next, the L-P map entries

containing modified selection bits are written back to their appropriate shadow locations. Once this is complete the new stable state is secured in stable memory but will not be used until a final atomic operation has completed. This final operation must be indivisible – that is it either completes successfully or not at all.

The final atomic operation is implemented by swapping the roles of two header pages as described by Challis [Challis 1978]. From the header page, the current and shadow versions of all the stable store data structures can be found. Each header page fits within a single physical disk block and therefore can only be written to disk as a single hardware operation. Each header page is time-stamped; thus, it is always possible to determine which header is the most up-to-date. The time stamp is written at the beginning and end of each header. If the two time-stamps disagree, the system can conclude that the header block is corrupt and elect to use the other header. The alternate will still describe a consistent view of the persistent store, but it will be one that represents the state of the system at the conclusion of the last successful stabilisation. Thus, the system will survive failure even during the final commitment of the stable structures.

It is possible for several independent stabilisation operations to be in progress at any time. Consequently, pages from more than one stabilisation may be written to the stable store concurrently. However, care must be taken to ensure that the stable store moves from one stable state to another in an atomic fashion. In practice, the final stages of stabilisation must be serialised. In particular, the L-P map can only make one flip at a time.

4. Cache Coherency

4.1. Communication architectures

Previous designs for coherency of shared persistent data have focussed on tightly coupled hardware. Such designs make use of the low cost of broadcast messages and snooping on bus traffic to implement coherency. Our design is aimed at distribution over a local area network where both broadcast and snooping are costly.

The protocol can be characterised as a central directory, multiple readers, single writer protocol. The design assumes that communication is reliable and that ordering of messages is preserved over point to point links. This assumption is safe when implemented using the Mach IPC abstraction.

The general aim of the protocol is to allow multiple clients to read the most up-to-date copy of a page, or a single client to write to the page without compromising the coherency of the pages. All read and write requests are made directly to the Stable Store Server. If a page has been modified since the last stabilisation and a current copy is not available in the store, read requests for the page are forwarded to a client with an up-to-date copy of that page. The server only services requests itself when it holds a valid copy of the page. Thus, up-to-date page copies may be distributed among the clients and the Stable Store. The aim is to maximise the freedom with which a client process is able to run, and prevent the server from becoming a bottle-neck for page retrieval and supply.

The Stable Store Server maintains all the information concerning the distribution and modification of pages. A data structure known as the *V-list* contains the identity of all clients holding a valid copy of a page. Similarly, the dependency list, or *D-list*, records which clients hold or have held a modified copy of a page since they last stabilised. If a client wishes to modify a page, it must already have read access to that page. If the page is not shared, a client may freely modify the page, although it must inform the Stable Store Server of the modification if the page was previously unmodified; the client does not require an acknowledgment before proceeding with modification of unshared pages.

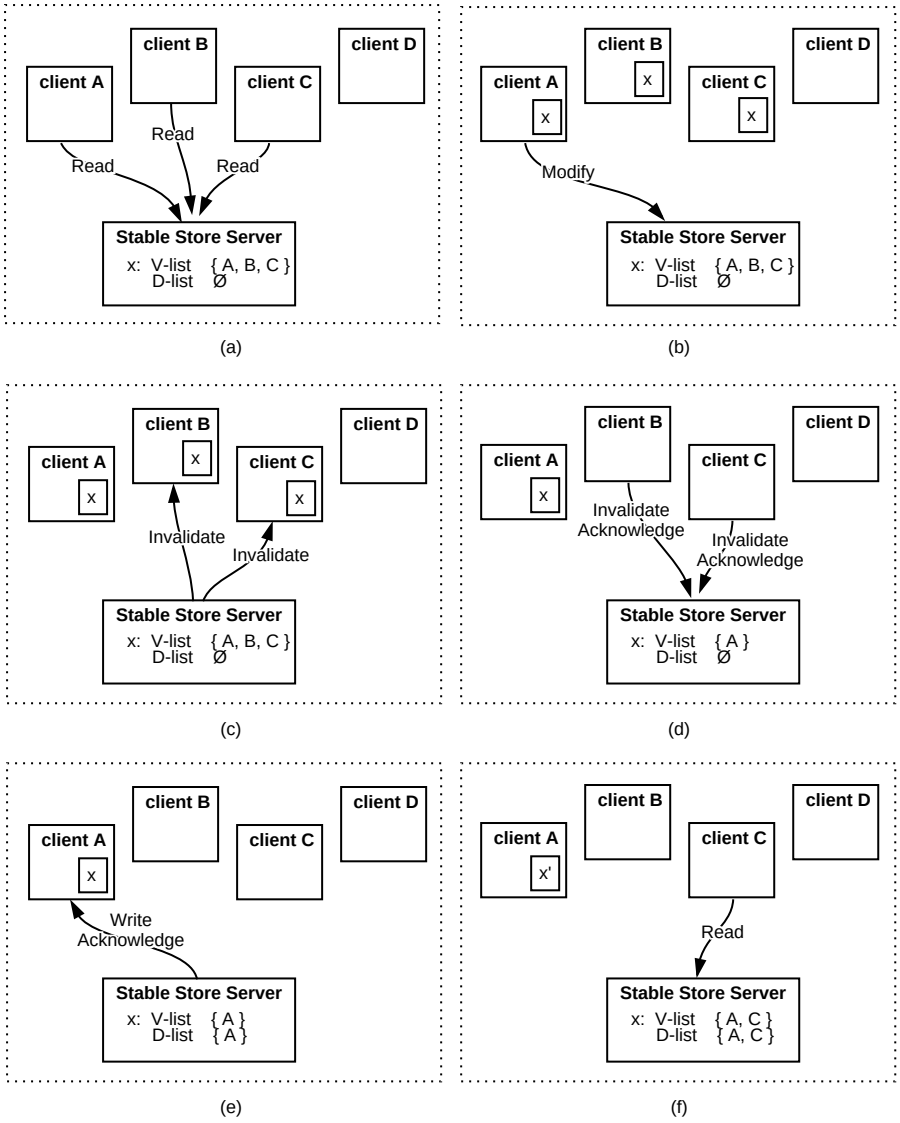

Figure 3. The modification protocol for a shared page.

These concepts can be illustrated as shown in Figure 3, which depicts the events involved in the modification of a page. As a result of each of the clients A, B and C attempting to read the page x, they have all been added to the V-list for that page; this is shown in Figure 3(a). Client A is attempting to modify the page and must forward to the Stable Store Server a modification request (Modify signal), as shown in Figure 3(b). The Stable Store Server next instructs all other clients with an up-to-date copy of the page to invalidate their copy; the identities of the clients to be notified are found from the V-list for the page, as shown in Figure 3(c). These clients must reply with an acknowledgment to the Stable Store Server on completion of the invalidation (Invalidate Acknowledge signal shown in Figure 3(d)). Upon receipt of an acknowledgment from a client that client is removed from the V-List for the affected page, recording that that client no longer holds a valid copy of the page. Once all acknowledgments are received the

client modifying the page is inserted into the pages D-list, recording that clients dependence upon the page. The Stable Store Server now sends a Write Acknowledge signal to the originally requesting client, as depicted in Figure 3(e), granting write permission. Any other client accessing the page results in that client receiving the up-to-date, modified page copy and being inserted into the V- and D-lists for that page.

The state to which a page belongs in both the server and the clients indicates which of the various properties are applicable to the page at that particular time. The states include: modified, shared, read requested (but not yet resident), valid copy held, modification requested (but not yet granted) and invalidation expected.

4.2. Maintenance of associations.

Associations are dynamic in nature and are constructed between stabilise cycles by the server. Associations may merge over time due to the sharing of data between previously independent associations.

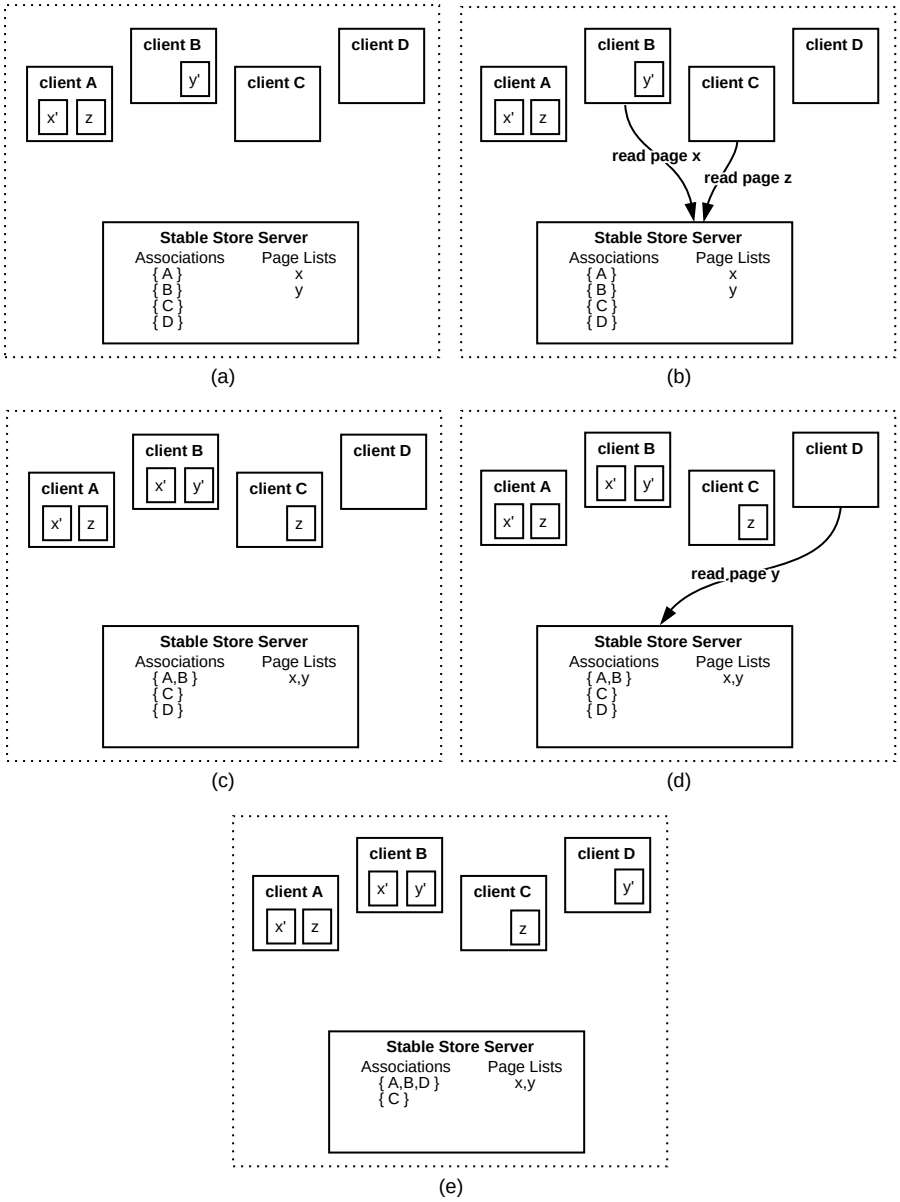

Figure 4. Expanding associations.

Figure 4 illustrates a combination of requests which lead to the expansion of an association. Firstly, Figure 4(a) depicts a collection of clients which do not share any pages; thus each association consists of precisely one client. Figure 4(a) also shows that client A modifies x and client B modifies y. Figure 4(b) depicts clients B and C each requesting pages held by client A; since client B is requesting page x, which has been modified by client A, the Stable Store Server merges the associations containing clients A and B. The association containing client C is not merged with this association, as the page requested (page z) is not modified. Thus, the failure or stabilisation of client A or C will not affect the other. The result of this operation is shown in Figure 4(c). A further request from client D for another modified page (page y) is shown in Figure 4(d); since

this page has been modified by client B, it causes the association containing client D to merge with that containing A and B. The result of this interaction is shown in Figure 4(e).

5. Clients

As shown in Figure 1, a client is divided into three main threads: the user program, the Client Request Handler and the external pager. Ideally, user programs should not be aware of the existence of the other parts of the client, perceiving only a single, flat, virtual address space. In reality, a few concessions must be made. The whole address range cannot be made available since a small area is required within which to place both the run time system and the coherency mechanisms. This area is reasonably small (a few megabytes) compared to the entire address space and is demand-paged by the default pager since it is not persistent.

The external pager handles any page faults or protection faults caused by the user program's attempts to access non-resident or protected pages. The Client Request Handler and the external pager jointly implement the client's part of the cache coherency protocol and are discussed in Section 5.1.

The Client Request Handler, described in Section 5.2, handles all incoming messages to the client from the Stable Store Server and from other clients. The cache coherency protocol requires that messages between particular pairs of communicants are delivered in the order in which they are generated. To maintain temporal ordering, all communications must be passed through the Client Request Handler.

As described earlier the system implements Napier88 operations atomically. Therefore the system must provide support for those operations which are not intrinsically atomic, such as multiple word reads or writes. This is discussed in Section 5.3.

5.1. External Pager

The abstraction of the persistent address space within a client is managed by the external pager. The coherency protocol requires the ability to be able to detect and service page faults and to selectively protect pages and handle attempts to violate those page protections within the persistent address space. The external pager provides this functionality.

The external pager is divided into two parts: a thread which fields requests from the kernel for maintenance of the persistent address space, and a routine library

which is used by the Client Request Handler to perform maintenance requests on the address space. The Client Request Handler maintains coherency; this may be as simple as changing local state information or may involve dialogue between the Client Request Handler and the Stable Store Server.

All protection exceptions and page faults caused by the user program's attempts to access pages are handled by the external pager. For example, when the coherency protocol requires notification of an attempt to modify a page, the page is protected against modification. Any subsequent attempt by the user program to modify the page will result in a page protection violation, which will be delivered to the external pager. The external pager will translate this into a Client Modification request and forward it to the Client Request Handler. In response to coherency management requests, the Client Request Handler will call the appropriate routine in the external pager interface, which replies to the kernel; this, in turn, reschedules the user program. The user program will retry and successfully execute the instruction which originally caused the exception or page fault.

The external pager also handles the return of modified pages to the Stable Store (to relieve pressure on local physical memory). If a removed page has been modified, an up-to-date copy must be returned to the Stable Store. If a removed page has not been modified, the Stable Store Server is notified that this client no longer holds a valid page copy.

In releases of Mach derived from Mach 2.5, the Mach kernel only informs the external pager of the removal of a locally modified page. In order to receive information on the removal of all pages (modified and unmodified), the external pager must ensure that all pages are modified (non-destructively) when they are brought into the client.

5.2. Client Request Handler

The Client Request Handler is the client's interface to the outside world. Some requests generated within a client are also passed to the Client Request Handler's port. The cache coherency protocol requires that messages between particular pairs of communicants are delivered in the order in which they are generated. To maintain temporal ordering, all external communications relating to a resident page must be passed through the Client Request Handler. The Client Request Handler is also responsible for maintaining the appropriate state information caused by external events via the coherency protocol.

The responsibility for start-up and creation of the other threads in the client rests with the Client Request Handler. It must also handle client termination and restart.

5.3. Atomic access

The Napier system requires that accesses to objects are atomic. This is necessary so that an object is never left in a partially modified, and hence inconsistent, state. If one process writes a value π into a location and another writes ε, either π or ε will always be read never a mixture of the two. This is true regardless of the type of π and ε. Most accesses are made to aligned 32-bit words, which is atomic at the machine level. However, there are some occasions when the atomicity provided at this level is insufficient. Such cases include accessing multiple word objects, such as real numbers (which are 64-bit quantities), bit-maps and discriminated unions. This requires a mechanism capable of providing user programs with atomic access to multiple data locations at arbitrary addresses.

In an unshared system it is possible to provide a lock on each object to which atomic access is needed. However this is undesirable when data is shared, since shared read only pages become modified when locks are set; voiding the gains made by though sharing. The Casper solution is to modify the cache coherency protocol so that it can be employed to deny access to pages that contain objects that are being accessed atomically.

This may be achieved using a structure which we call a *latch*. The semantics of a latch is analogous to a door latch: it may be set before the door is closed, but once the door is closed, the door will not open again until the latch is released.

Atomic access to multiple locations at arbitrary addresses may be implemented via latching each affected page. Two latches are provided per page – a read latch and a write latch. A latch, when set, prevents the release of the page to any other client for the purpose indicated by the kind of latch. If a page is required for an atomic read operation, the write latch is set and so a write operation by another client occurring part way through the read operation is prevented. If an atomic write is desired, the read latch is set to prevent the page from becoming shared part way through the write.

When the need arises to access more than one page, pages are latched serially and in ascending address order to prevent circular dependencies with competing clients, and hence avoid deadlock. Once all of the necessary pages are resident and have the appropriate latches set, the atomic access is performed and the latches released.

The design of the latching mechanism has aimed for efficiency, particularly for common cases, such as when only one page is needed and the page is already resident. Although shared by concurrently executing threads (the user program and the Client Request Handler) latches do not require protection from concurrent access since they are:

- only ever set by the user program,
- released by the user program or by the Client Request Handler (when the user program is guaranteed to be blocked), and,
- only read by the Client Request Handler.

Thus a request to latch a page only takes a few machine instructions and has minimal effect on execution speed.

5.4. Local Heap Management

Each client maintains a *local heap* for local object creation; this is a previously unused set of contiguous pages drawn from the persistent address space. Local heaps are small enough to always remain resident within the client's page cache during normal execution. Our 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 using this model. This can result in improved performance from better page fault behaviour and improved processor cache utilisation.

If transient objects are confined to a localised area, they may be garbage collected locally at low cost. Local heaps may be safely garbage collected provided that no external references (from other processes or the Stable Store Server) point into them. Fortunately, the creation and export of such pointers is easily detected, making this technique tractable. The Casper system utilises an algorithm [Koch, et al. 1990] based on generation-based garbage collection [Ungar 1984] to maintain the local heap.

6. Conclusion and future plans

The architecture described in this paper supports concurrent access to the persistent store by users on workstations connected by a local area network. This architecture consists of an arbitrary number of clients sharing a coherent persistent store managed by a central server. Despite the fact that this architecture is designed to support an object-based language, the coherency of the stable store is maintained at the page level. Many other persistent object architectures have

chosen to employ software object address translation when objects are moved from long term to short term memory. The motivation for this work was to experiment with the utilisation of hardware address translation mechanisms via the Mach external pager mechanism.

This approach has led to some complications, such as those discussed in Section 5. However, we believe that the benefits outweigh the disadvantages. One of these benefits is that our scheme exploits object clustering on pages within the persistent address space to reduce the frequency of requests to the Stable Store Server by providing a degree of prefetching.

Overall, the Mach operating system has proven to be an excellent platform for the conduct of this research. In particular, the use of external pagers to support programmable page fault handling is central to the implementation of cache coherency in Casper. Furthermore, the single inter-process communication (IPC) structure available in Mach permits a transparent interface between components of the architecture, independent of the physical location of the communicating parties. However, our work with Mach has highlighted some deficiencies in the current version of the operating system.

A significant inconvenience is the kernel's removal of pages according to its own LRU algorithm. It would be more useful if the kernel requested the external pager to remove one or more pages, rather than sending its own choice of pages to the external pager for removal. This is due to the fact that the pages selected may contain pointers into the client's local heap area, in which case removal is a costly operation in our system. The external pager can determine more appropriate candidates for efficient page removal through the available state information.

The architecture described in this paper is an experimental framework for further investigations. These experiments will focus on three areas: garbage collection algorithms, the utilisation of pointer swizzling to implement very large address spaces and clustering algorithms. At the time of writing, Casper is running under Mach 2.5 on a Sun 3/60.

Acknowledgments

This work was partly supported by Australian Research Council grant number 4900-6830-1000. The work described in this paper has been carried out by a research group at the University of Adelaide; apart from the authors, other members of this group to make significant contributions towards the design and

implementation of the system include Alex Farkas, Ruth Fazakerley and Bett Koch. We also acknowledge the continuing cooperation which we have received from the Persistence Project at the University of St Andrews.

References

Acceta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. & Young, M., "Mach: A New Kernel Foundation for Unix Development", Proceedings, Summer Usenix Conference, pp. 93-112, 1986.

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

Atkinson, M. P. & Morrison, R., "Procedures as Persistent Data Objects", Transactions on Programming Languages and Systems, 7(4) pp. 539-559, 1985.

Atkinson, M. P., Morrison, R. & Pratten, G. D., "Designing a Persistent Information Space Architecture", *10th. I.F.I.P. World Congre*ss (North-Holland, Amsterdam), pp. 115–120, 1986.

Bancilhon, F. & Maier, D., "Multilanguage Object-oriented systems: New answers to old database problems", in *Future Generation Computers II*, Fuchi, K. and Kotti, L. Eds. (North Holland, 1989).

Berry, D., "Block Structure: Retention or Deletion?", Third Annual ACM Symposium on the Theory of Computing pp. 86-100, 1971.

Cardelli, L. & Wegner, P., "On Understanding Types, Data Abstraction and Polymorphism", Brown University, 1985.

Challis, M. F., "Database Consistency and Integrity in a Multi-User Environment", in *Databases: Improving Useability and Responsiveness*, (Academic Press, 1978).

Dearle, A., "Environments: A Flexible Binding Mechanism to Support System Evolution", Proc. 22nd Hawaii International Conference on System Sciences pp. 46-55, 1989.

Eswaran, K. P., Gray, J. N., Lorie, R. A. & Traiger, I. L., "The Notions of Consistency and Predicate Locks in Database Systems", Communications ACM, 19(11) pp. 624–633, 1976.

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

Koch, B., Schunke, T., Dearle, A., Vaughan, F., Marlin, C., Fazakerley, R. & Barter, C., “Cache Coherence and Storage Management in a Persistent Object System”, Proceedings, The Fourth International Workshop on Persistent Object Systems (Morgan Kaufmann), pp. 99-109, 1990.

Lampson, B., “Atomic Transactions”, in *Lecture Notes in Computer Science Vol 105: Distributed Systems – Architecture and Implementation*, B., L., M., P. and H.J., S. Eds. (Springer-Verlag, Berlin, 1981).

Lorie, R. A., “Physical Integrity in a Large Segmented Database”, Association for Computing Machinery Transactions on Database Systems, 2(1) pp. 91-104, 1977.

Morrison, R., Brown, A. L., Connor, R. C. H. & Dearle, A., “The Napier88 Reference Manual”, University of St Andrews, 1989.

Philipson, L., Nilsson, B. & Breidegard, B., “A Communication Structure for a Multiprocessor Computer with Distributed Global Memory”, 10th International Symposium on Computer Architecture pp. 334-340, 1983.

Ungar, D., “Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm”, ACM Sigsoft/Sigplan notices, 19(5) pp. 157-167, 1984.

Wu, K.-L. & Fuchs, W. K., “Recoverable Distributed Shared Virtual Memory”, IEEE Transactions on Computers, 39(4) pp. 460 - 469, 1990.