A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems *

Ming-Chit Tam
Jonathan M. Smith
David J. Farber

Distributed Systems Laboratory
Dept. CIS, University of Pennsylvania
Philadelphia, PA 19104-6389

May 15, 1990

Abstract

Two possible modes of Input/Output (I/O) are "sequential" and "random-access", and there is an extremely strong conceptual link between I/O and communication. Sequential communication, typified in the I/O setting by magnetic tape, is typified in the communication setting by a **stream**, e.g., a UNIX[1] pipe. Random-access communication, typified in the I/O setting by a drum or disk device, is typified in the communication setting by **shared memory**. In this paper, we study and survey the extension of the random-access model to distributed computer systems.

A **Distributed Shared Memory (DSM)** is a memory area shared by processes running on computers connected by a network. DSM provides direct system support of the shared memory programming model. When assisted by hardware, it can also provide a low-overhead interprocess communication (IPC) mechanism to software. Shared pages are migrated on demand between the hosts. Since computer network latency is typically much larger than that of a shared bus, caching in DSM is necessary for performance. We use caching and issues such as address space structure and page replacement schemes to define a taxonomy. Based on the taxonomy we examine three DSM efforts in detail, namely: IVY, Clouds and MemNet.

*This research was supported by Bell Communications Research, Inc. under Project DAWN
[1]UNIX is a registered trademark of AT&T Bell Laboratories.

1 The Distributed Shared Memory Concept

A Distributed Shared Memory (DSM) is a memory space that is logically shared by processes running on computers connected by a communication network. While such an organization exists in shared memory multiprocessors, in the domain of distributed systems it is unusual. Most existing distributed systems [Tanenbaum 85] are structured as a number of processes with independent address spaces. These processes communicate with each other through some form of interprocess communication (IPC), typically message passing or remote procedure call. In a DSM system, data sharing (and thus IPC) is supported directly. Processes communicate with each other by reading and modifying shared directly-addressable data. A DSM can be a flat and paged virtual address space [Li 86], a segmented single level store [Ram 88], or even a physical address space [Delp 88].

This paper is a survey of some current research efforts on DSM. Section 1 elaborates on the concept of DSM, motivations for DSM, potential advantages, and research issues. The section concludes with a brief overview of the systems we have chosen to examine. Sections 2, 3, and 4 are detailed discussions of three implementations, IVY[Li 86], Clouds[Ram 88] and MemNet[Delp 88]. Section 5 compares their approaches. Section 6 gives a short review of some current research efforts in DSM. Section 7 concludes the paper and suggests directions for future work.

1.1 Why do we want DSM ?

A distributed system can be viewed as group of computers cooperating with each other to achieve some goal. These computers are autonomous, in that each computer has an independent flow of control, and there is no sharing of physical memory between them, unlike multiprocessors. Processes running on different computers have distinct address spaces. They communicate by sending and receiving messages. An important characteristic of cooperation is state sharing[Cheriton 86]. Unfortunately, message passing primitives do not support data sharing directly. Data sharing is still possible with these primitives. This can be done by implementing the shared data in a dedicated process and operating on the data by sending predefined operations to this process[Libes 85]. Other methods may involve moving data around explicitly using message passing primitives. Special care must be taken to maintain the consistency if a piece of data is replicated.

As more experience is gained with message passing programming, it is found that having to move data back and forth explicitly within programs puts a significant burden on application programmers. Remote procedure call (RPC) [Birrell 84], was introduced to provide a procedure call like interface. Since the "procedure call" is performed in a separate address space, it is difficult for the caller to pass context related data or complicated data structures, i.e., parameters must be passed by value. Birrell indicated the desire for distributed shared memory so that data could be passed by reference. RPC can be viewed as a "poor man's" version of shared memory, since the semantics are basically those of shared memory, with limitations imposed by implementation constraints (e.g., limited copying of data).

A shared memory space provides direct support for data sharing. The mapping of shared data to a shared memory space is natural; Young, et al [Young 87] have observed the relationship between memory and communication. Thus, the question of extension to a distributed setting arose. Ideally, processes on each node should be able to access the same address space with fetch and store operations. However, since the latency involved in communication through the network is high, simple implementation of the fetch and store as remote operations to a shared memory server is not attractive[Spector 82]. "Latency" represents a speed ratio between remote access and local access, and if the value of this ratio is large, the mismatch must be remedied for adequate performance. Such a mismatch, albeit a generally smaller one, exists in shared memory multiprocessors. Thus, we look to shared memory multiprocessor architectures for inspiration.

1.2 Shared Memory Machines

Shared memory multiprocessors speed up a computation by sharing the data and operating on them in parallel. Sharing is achieved by implementing the data in a shared memory space addressable by all the processes in the computation. Processors and memories are interconnected through a shared bus, which permits global addressability.

However, the main memory may be too slow for the powerful processors. Moreover, as we add more processors to the system the traffic on the shared bus becomes heavy, causing serious delay on fetch and store operations. Caching is a potential solution to the above problems. Unfortunately, multiple processor caching may cause several copies of data to coexist in different caches. When the data is changed, we run into the danger of reading an old copy. A **cache coherence protocol** is needed to ensure that we will always read a valid copy. This usually involves invalidating all the copies or updating all of them [Arch 86] whenever there is a write. Since a cache is of finite size, it is also important to have a cache replacement policy.

The problem of maintaining cache coherency has been studied extensively in the design of parallel computer architectures. There are two basic approaches, namely the "snooping cache" and "directory based" approaches.

1.2.1 The Snooping Cache Approach

The snooping cache approach relies on the existence of some communication medium with a broadcasting capability, e.g., a shared bus. Each cache is required to monitor the shared bus for memory transactions initiated by other processors to maintain the coherency of its own data. A good example is the Berkeley protocol[Arch 86].

A Snooping Cache Example - Berkeley Protocol

The Berkeley protocol assumes a physical shared memory accessed through a single bus. A block can be in one of four states, namely dirty, shared dirty, valid and invalid. The protocol adopts an ownership scheme, the owner of a block is either a cache or the memory. An owner is the last entity that modifies the data. If the block is flushed by the owner, then the main memory is designated as the owner.

A block in the dirty state can not be shared, i.e., it is in residence in only one cache. A block in the shared dirty state may have duplicates in other caches. Both states represent the ownership of the block. When there is a read miss, the block is supplied by its owner. If it happens to come from a host with exclusive access to it, then the status of the block at that host changes from dirty to shared dirty. The cache of the faulting process will get the block and mark it valid. Valid and invalid blocks can simply be discarded on replacement, but dirty and shared dirty blocks must be written back to the main memory.

A write can proceed if the destination block is cached in a dirty state. On the other hand, if the block is in the shared dirty, valid or invalid states, then an invalidation signal must be sent to other caches. In the latter case, the current owner must also return the block as well.

Another strategy is to broadcast the update to all the caches whenever a write take place. This protocol makes writes expensive and economizes on reads. The Berkeley protocol makes writes less costly at the expense of invalidating read-only copies. The snooping cache approach is limited in scalability due to its reliance on the shared bus. Moreover, normal traffic between the processor and its cache can incur delays from checking the cache on every memory operation monitored on the bus.

1.2.2 Directory Based Scheme

The directory based approach addresses the scalability problem by putting a directory of memory blocks in the main memory. Whenever a cache miss occurs, the request is first directed to this directory. There are many variants of this scheme; see [Agarwal 88] for a detailed discussion. Typically, each entry in the directory includes the ownership, the copyset, and a dirty bit for the block. The copyset contains information on which caches have a copy of the block; this copyset can be implemented as a bit vector. When a read miss occurs, the dirty bit in the directory is examined. If the block is not dirty, then the version in the main memory is valid and the block is simply returned with the copyset information updated. If the dirty bit is set, then the owner of the block must have modified the block. It is necessary to update the version at the main memory, and when this is done, the read copy is supplied. A write miss or a change from read permissions to write permissions requires the copyset information from the directory to invalidate the other copies. Unlike the snooping cache scheme, the location of read copies is well-known. Hence, it is possible to send the invalidation sequentially rather than broadcasting it. The directory scheme does not require a broadcast medium, but it does need an extra lookup on every cache miss.

Most of the distributed shared memory systems use variants of the above protocols. IVY and Clouds used the directory scheme while MemNet used the snooping cache scheme.

1.3 General Architecture of Distributed Shared Memory Systems

Since caching is a good solution to address memory access latency, it could also be a good solution to network latency. All the distributed shared memory systems implement caching, to bring the expected access time near to that of local memory. Most of them use the main memory of the hosts to cache pieces of the shared address space. If we call the basic unit of caching a page, then the scenario is that of pages migrating from one host to another on demand. A cache coherence protocol is followed to ensure the consistency of the copies.

1.4 Issues of Distributed Shared Memory

With the general architecture of DSM in mind, we examine the issues involved in implementing this architecture. These issues, when addressed, define a taxonomy for our subsequent discussions in this paper.

1. **Structure of the shared address space** - A distributed shared memory is just a shared address space structure. The structure of the address space is dependent on the type of applications that distributed shared memory is intended to support. The address space can be *flat*, *segmented* or *physical*.
2. **Cache coherence protocol** - Since different cache coherence protocols make different assumptions and tradeoffs, the choice is dependent on the pattern of memory access and also the environmental support. For example, the latency of network communication can make a cache miss expensive. Hence choosing the right coherence protocol is important.
3. **Synchronization Primitives** - A cache coherence protocol alone cannot maintain the consistency of shared data when we have concurrent accesses. We need synchronization primitives to synchronize the access of shared data, e.g, semaphore, eventcount and lock. It is important for a distributed shared memory system to provide such primitives.
4. **Block Size** - The block size of a cache is an interesting parameter. It depends on the cost of communication and the type of locality exhibited in the application, etc. Block size is usually a measure of the granularity of parallelism explored. All our examples have well justified reasons to support their choice of block size.
5. **Replacement Policy** - Finally, since there is only a limited amount of shared memory at each node, there is always a possibility of cache overflow. Thus distributed shared memory system must have strategies for cache replacement and some form of backing store.

Most of the above issues are addressed in the examples of DSM to be described. Varying degrees of attention were spent on issues depending on the designer's focus.

1.5 Our Examples

While [Libes 85] discussed shared variables addressed through a procedural interface in 1985, what we consider true distributed shared memory systems did not appear until 1986. These early systems were IVY[Li 86] and MemNet[Delp 86]. IVY is a software implementation of DSM and MemNet is a hardware implementation of DSM. IVY covered many of the semantic issues in DSM, as well as addressing protocols. The objective of MemNet was different, it attempted to prove the notion that shared memory paradigm [Farber 88] can shorten the communication software path. However, in this proof, it addressed many of the issues IVY did.

Clouds [Ram 88] adopted the DSM notion to provide direct support of object mobility. Its major contribution involves combining the cache coherence protocol with the synchronization of shared data access to give a more efficient implementation of DSM. These are the three examples that we will use in our discussion. Section 6 gives a short review of other DSM work.

Earlier in this section, we explained the motivations behind DSMs. We have also described the general architecture of a DSM system and the issues that a DSM system

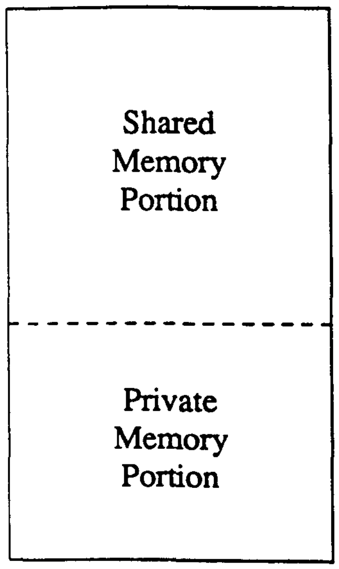

Figure 1: Process Address Spaces in IVY

must address. DSM provides direct support to the shared memory style of programming and allows sharing of complicated data structure across machine boundaries, thus making these boundaries transparent. When augmented by hardware it provides us with a short cut through the host communication software path.

2 IVY - Distributed Shared Memory in Software

IVY is implemented on the Apollo Domain Architecture[Nel 84]. It investigated the feasibility of providing a virtual shared memory environment on loosely coupled multiprocessors. It was targeted towards applications suited to parallel processing.

In IVY, a process address space is divided into private and shared portions. The private portion is not addressable by other processes and the shared portion is implemented as a virtual shared memory. A virtual shared memory is a flat address space shared by all the processes running on different nodes, i.e., a single address space shared by threads. This mode of sharing is different from that used in Multics[Daley 68][Bensoussan 72] where address aliasing is used. Figure 1 shows the structure of an IVY address space.

The address space is paged. A page is the minimum unit of synchronization; it migrates from one node to another on demand. Like all systems that use caching, IVY made the assumption that programs exhibit locality. Once a process has paged in its working set, it will concentrate references on this set for a period of time. Part of the main memory of each station is dedicated to cache the pages with the disk used as the backing store. There is a memory manager at each node to satisfy both local and remote requests and implementing the cache coherence protocol. When a reference to an address in the shared space is generated, the faulting process is blocked and the IVY memory manager checks if the page is local. If the page is absent, then a remote memory request is made. When the page is acquired, the process generating the page fault is resumed.

The advantages of having shared virtual memory include the direct support of the shared memory programming paradigm, the ability to pass complicated data structures between processes, and the ease of process migration. The first two advantages have already been explained in section one. Process migration has been shown to be tedious[Smith 88]. Virtual shared memory provides good support for process migration because it allows the migrated process to demand its pages, or at least the subset resident in the DSM, from the previous processor. Hence process migration could be as operationally simple as attaching the process control block (PCB) to the ready queue of a remote processor.

We shall now look at IVY more closely by first looking at its cache coherence protocol followed by some of the memory management issues and finally the implementation of **event count** as its synchronization primitive.

2.1 The Coherence Protocol

The notion of coherence used in IVY is a multiple reader/ single writer semantics. A read operation on a particular address will always get the last value written to that address. It is up to the coherence protocol to enforce this semantics. As mentioned before, cache coherence protocols are a well-explored topic in the field of closely coupled multiprocessers. The typical "write-through" update scheme is unsuitable for distributed shared memory since it would require a network access on every write operation. So the possibility of using an invalidation scheme was explored. However, distributed systems are distributed and asynchronous in nature, so there are some differences. We look at these factors in the next section.

2.1.1 Applying Multiprocessor Protocol in a Distributed Environment

In a closely coupled system, since all the caches are connected by a bus, whenever there is a cache miss all the caches are notified and the owner can respond accordingly. The owner of a page is the last host modifying the page. In a distributed environment we may not always have this luxury. Hence the location of the owner of a page becomes an important issue. In closely coupled systems, invalidation is done via broadcasting. However, in non-broadcast networks this could be expensive. Moreover, in closely coupled systems using the snooping cache scheme, the memory request cycle is atomic, i.e., the generation of the block miss signal, the response of the owner, and the invalidation of the other blocks are done synchronously. In a distributed environment, without an acknowledgement it is impossible to tell whether a remote processor has done the invalidation This summarizes the difference between the two environments. Following the above argument, a directory based scheme with acknowledgement to invalidations becomes a natural choice.

2.1.2 An Overview of the Protocol

Each host in IVY has its own page table; each entry of the page table records the access rights of the host. A host may have read, write, or no right to a page. The access right of a page is equivalent to the state of a block in a cache. The table below shows the equivalence.

Cache	*IVY*
dirty	write
shared dirty	owned read
valid	read
invalid	nil

When a shared address is referenced, the host checks whether it has the right to access the page containing the address in the specified mode. If this is not the case, then either a read or write fault is generated depending on the mode of access. Faults are handled as follows :

Read Fault :

1. find out who is the owner.
2. owner add the faulting host to the copyset.
3. owner change its access right to read only.
4. owner send the page to the faulting host.

Write Fault :

1. find out who is the owner.
2. owner sends the page and its copy set to the faulting host and mark its entry invalid.
3. the faulting host sends out invalidations based on the copyset.
4. the acknowledgements to the invalidations come back and the process proceeds.

In both cases, the first step is to find the owner of the page. The **copyset** is the set of hosts that has a read only copy of the page. It allows us to avoid the need for broadcasting the invalidations.

We will describe three different coherence protocols proposed. These protocols differ mainly in the way they implement the directory and hence locating the owner of a page. The first one is a centralized scheme, the second one is a distributed partitioned scheme and the third one is a dynamic scheme. In all the schemes there is a page table in each host. Each page entry contains at least the access right, the physical location, a copyset and a lock. The lock is used to synchronize the access to the page table entry. This is particularly useful when there is a fault on the page as the lock will prevent multiple faults from processes in the same host and will hold the incoming requests from the network.

2.1.3 Centralized Scheme

In the centralized scheme, there is a central page manager whose central table keeps track of the locations of all the pages. The identity of the manager is well known to all the hosts. When there is a read fault , the faulting host sends a read request to the central page manager who would forward the request to the owner of the page. The owner adds the faulting host to the copyset of the page and send the page back. If the request is a write, the central table must also be modified so that the owner of the page becomes the faulting host.

The scheme requires two messages to locate the owner and one message to pass back the page from the owner. Write requires a number of messages equal to the size of the copyset to invalidate the copies. The problem with the centralized approach is that the host running the page manager may become the bottleneck of the system.

2.1.4 Fixed Distributed Scheme

The fixed distributed scheme is a direct extension of the centralized scheme. It avoids the bottleneck problem by distributing the role of the central manager. Every processor is given a predetermined subset of pages to manage. The mapping from pages to processors is described by a mapping function. Whenever a page fault occurs, the mapping function is consulted and the page request is sent to its page manager. The message overhead is close to that of the centralized scheme.

In the above schemes, the location of a page is always kept by a page manager. Concurrent requests are serialized at the page manager holding the location. It is possible to eliminate the page manager altogether by having each host keep track of the pages.

2.1.5 A Dynamic Distributed Scheme

In this scheme the page managers are eliminated, and the page table entry in each host is extended by having an additional attribute called probowner (probable owner). This attribute gives the host a hint on the location of the owner. If a host receives a request on a page where it is not the owner it forwards the request according to the hint from its page table.

The hints are updated under the following conditions :

1. a host receives an invalidation request
2. a host relinquishes ownership, i.e., on write fault
3. a host receives a page
4. a host forwards a page fault request

When a host (1) receives an invalidation request, it knows that there must be a transfer of ownership and so it must make the change. (2) is obvious. When a host (3) receives a page for write, it becomes the owner of the page. When it receives a page for read, it would also know who is the true owner. Finally when a host (4) forwards a page request, if the request is for write, then the faulting host is going to be the new owner. If the request is a read, then we know that after the request is satisfied, the faulting host will have the correct ownership information. In either case, it is a good idea to change the ownership information of the page to the faulting host.

It was proved [Fowler 86] that the algorithm will always terminate by finding the true owner of the page. A similar scheme was applied to finding a migrated message recipient in the DEMOS/MP operating system [Powell 83]. In the worst case, the dynamic scheme may take N-1 messages to locate the owner of a page. However, due to the effect of hint update, the overhead is much better than N-1 on average. The worst case overhead of locating K owners of a page in a system of N hosts where only p of the hosts are interested in the page is O(N + Klog p). This means that if our system has N hosts and only p of them share the page, then the overhead in page location for K successive write faults is of order of N + Klog p. The dynamic scheme provides us with a distributed way to locate owners of pages. On average we still need at least log p message to find the host. Li improves on this by performing periodic broadcasts to keep the location information current.

The initial condition is simple, we set the probowner field of each entry to be a particular host to which we give it the ownership of the page.

2.2 Memory Management

In this section we shall look at some memory management issues of virtual shared memory and in particular, investigate why they are so different from conventional memory management. We will first look at the page replacement problem and then the memory allocation problem.

2.2.1 Page Replacement

Since the size of the physical memory in any machine is always limited (usually much smaller than the virtual address space), it is important to have a page replacement scheme [Peterson 86]. Traditional page replacement policies like LRU cannot be applied directly to IVY's virtual shared memory.

There are five kinds of pages in IVY's virtual shared memory, namely writable, read-owned, read only, nil and unused. A page frame in a nil state is one whose corresponding virtual page was invalidated. Both nil and unused pages have the highest replacement priority, i.e., they will be replaced first if a page is needed. It is obvious for unused page. For nil page, since the corresponding virtual page has already been invalidated, future access to the page would cause a page fault and so the current content is not useful anyway. Notice that a nil page may be one that is referenced recently, this is exactly why a simple LRU method is not adequate. The read only pages have the next highest priority. Since a read only page would be backed up by its owner, it is possible to simply discard that page. However, if the host may require the page in the future then the page must be brought back from the owner and network access is involved. If we had written the page onto a local disk, the network access could have been avoided. Obviously, the tradeoff is on how likely is the page going to be read in the future. If the page replacement must be performed on a remote page server, we should just discard the page. Read owned and write owned pages are paged to disk.

In addition to replacement of pages using secondary store, it is possible to make use of the main memory of other nodes. As before, nil and unsed pages are simply discarded. Read only pages are also discarded. For read-owned pages and writable pages, discarding them would certainly require transfer of ownership. If a page is read-owned, we can avoid transferring the whole page by finding a host that has a copy of the page. This can be done by using the copyset information. Writable pages or read owned pages with no copy available are replaced by finding the host that has the maximum number of pages in nil access mode or read only mode. This may require each node to keep a table about the state of memory allocation in other nodes. The page replacement algorithm chooses the node by consulting this table. The table is updated by having each node piggyback its memory information during normal traffic.

While the replacement algorithm described above prioritizes the pages according to the their state, it is possible to add in the last referenced time as an additional parameter. This gives us a LRU replacement policy with classes.

2.2.2 Memory Allocation

To support dynamic data structures, it is essential to have a dynamic memory allocation scheme. IVY had explored two ways to do dynamic memory allocation. The first one is a centralized approach where all processes request memory from, and deallocate memory to, the centralized memory allocator. The centralized allocator is a simplistic solution. The second approach is two level, where each node has its own allocator routine in addition to the centralized one. Each node would request a large chunk of memory and administer the memory requests from local processes. The centralized manager is contacted only when a local node is running short of memory. A local allocator may deallocate memory to the centralized allocator explicitly or wait until the centralized routine requests more. There is a clear trade off here between the number of messages and the efficiency of system memory management.

2.2.3 Summary

In this section we have examined some of the memory management issues of virtual shared memory. These issues are similar to those in conventional virtual memory. However, due to the unique characteristics of virtual memory (like nil page and read only page) and also to its distributed nature, solutions to these issues must be modified in some ways. In the case of the replacement problem, a redefinition of replacement priority is necessary and with dynamic memory allocation, mutual exclusion becomes necessary.

2.3 Process Synchronization

The cache coherence protocol prevents the existence of inconsistent copies in the cache. However, it does not guarantee serializability during concurrent access, e.g, the concurrent execution of two assignment statements for the same location. Hence we need primitives like semaphores or eventcounts to synchronize accesses to shared variables.

IVY chose to implement eventcount[Reed 79] as the basic synchronization primitive. An eventcount records the number of occurrences of a particular event, e.g, access to a shared variable. A process waits for the nth occurrence of an event by executing **wait(eventcount,n)** and signals the occurrence of an event by **advance(eventcount)**. A **read(eventcount)** operation is also provided to read the current value of an eventcount.

It is possible to implement an eventcount using distributed shared memory and using the coherency protocol to support multiple copies. However, the need to modify the process queue associated with an eventcount means that some lock must be provided to avoid simultaneous access. This lock can be provided by implementing it in the shared memory and operating on it using test and set instruction only. However, this could cause many faults if many processes are interested in an eventcount. Hence, eventcount is implemented using the RPC independent of the shared memory. A simple scheme is to put all eventcounts onto a single node. A fixed distributed scheme is also possible and it avoids the potential bottleneck in centralized scheme.

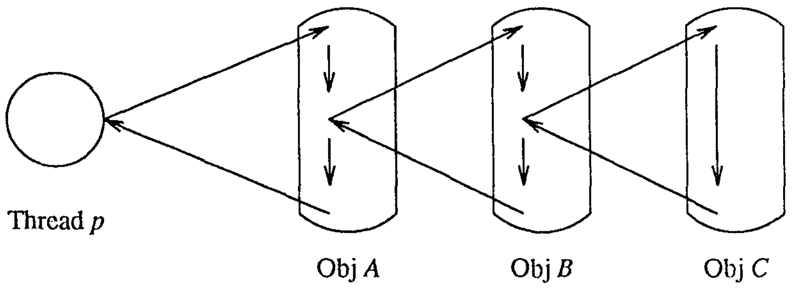

Figure 2: Object Invocation Model of Clouds

Li experimented with several programs on IVY, each of them with different computation and communication characteristics. In general, parallel programs with heavy computational needs and minimal access to global data usually perform better. This is not surprising, as such programs place the least burden on the DSM mechanism. In addition, the large number of instructions executed by the computationally intensive programs reduce the relative overhead of DSM support. DSM allows one to use the main memory of all the participating machines. Hence, when the amount of data is large, DSM might allow improvement over a large-memoried uniprocessor in a superlinear manner as the uniprocessor may devote a larger time to page fault service.

3 Clouds - Supporting Objected Oriented System Using DSM

3.1 An Overview of Clouds/Ra

Clouds[Das 88] is a single level store object oriented system. The computation model of Clouds consists of passive objects with threads.

Threads are the flows of control. During the computation, a thread executes within a Clouds object. It travels from one object to another through object invocation. Each object has its own address space and is installed as the address space of the thread when it is invoked. A remote object invocation may either be implemented as a remote procedure call or as a movement of data. When an invocation on a remote object occurs, the object is first located. In the former case a process is constructed on behalf of the thread at the remote node and the execution started. Finally the result is passed back to the calling thread. In the latter case, we bring the whole object to the node of the calling thread to execute the invocation. Since the address space of an object is composed of segments, an efficient way to move and share segments is required to support the object mobility. Distributed shared memory provides good support.

3.1.1 Ra - The Minimal Closed Kernel

The kernel of the Clouds operating system is known as Ra[Auban 87]. Ra is minimal in the sense that it only provides the necessary support for system services, e.g., object invocation mechanism. It is closed because changes in system services do not involve modification of the kernel. Ra supports several primitive abstractions, **segment, isiba and virtual spaces**. An object is implemented by an object virtual space together with an invocation mechanism. A virtual space consists of a number of segments and the mapping of the virtual space to the segments is described in a segment called the virtual space descriptor. A segment is a data container, it is the most elementary level of abstraction. Each segment belongs to a system object called partition and has a unique system wide id. It is the partition that implements the storage of segments. In this sense, a partition resembles the external page manager of Mach (mentioned below in section 6); it abstracts the physical storage of segments. Structuring an object as a collection of segments allows objects to share segments, e.g., code and templates. This is important when implementing concepts such as inheritance.

Isiba is the basic unit of computation, it consists of a context segment and a stack segment. Each isiba has two address spaces, namely the **p space** and also the **o space**. When the p space of an isiba is instantiated with a process virtual space, the isiba becomes a process. A process virtual space typically consists of a context segment and stack segments, one for each object invocation. An isiba with no process virtual space acts as a system demon. The o space of an isiba is the virtual space of the currently invoked object. A Clouds thread is implemented as a number of processes across a group of machines with one process for each machine. Figure 3 shows the computation and the storage hierarchies in Clouds.

3.1.2 The Object Invocation Mechanism

An object invocation involves a call to a system object called the invocation manager. The invocation call includes the name of the calling object, the name of the object to be invoked, the operation that is to be performed and also the parameters of this operation. The invocation manager maintains a table about all the local objects. If the invoked object is local, the virtual space of the object is installed into the object space of the isiba. Otherwise, the invocation manager will ask for help from a locator object. When the object is located, the system may either spawn a stub process at the remote site or bring the object to the local site. In our case, we are only interested in the latter.

An object is installed by fetching its virtual space descriptor. Since the descriptor contains information about the mapping from virtual space to the segments, we can start executing at the entry point of the invoked operation when the virtual space is installed. Segments are then be paged in as required.

3.2 Distributed Shared Memory

Since objects are allowed to migrate from one node to the others, segments must likewise be migratable. To provide efficient invocation mechanisms, we need a uniform way to support the migration and sharing of segments. By viewing segment space as a shared memory cached by nodes, we get distributed shared memory [Ram 88].

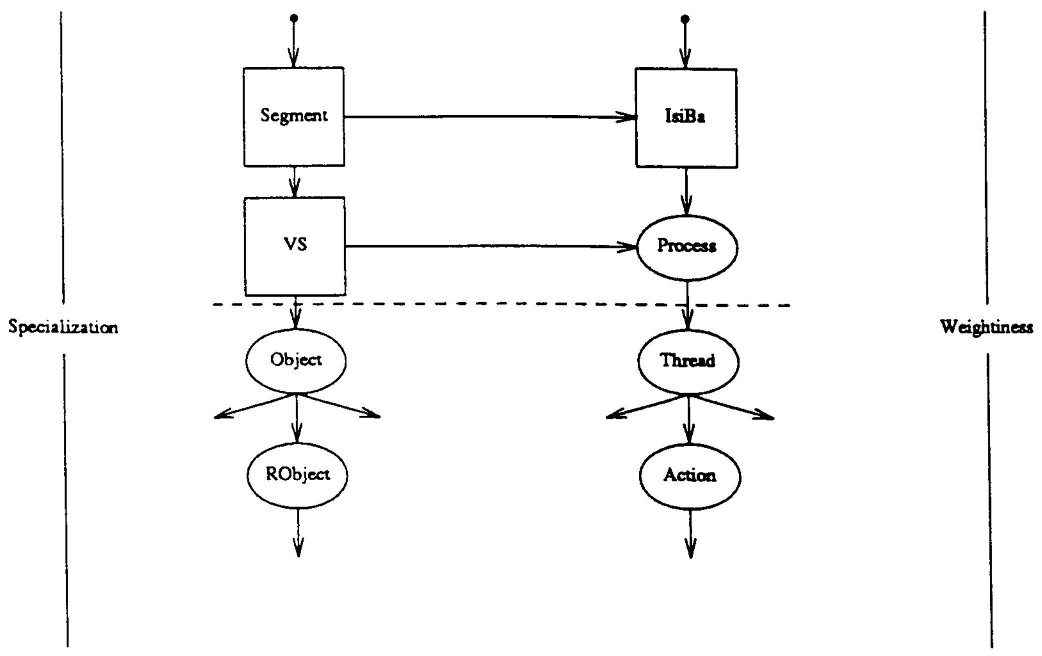

Figure 3: Computation and Storage Hierarchies of Clouds

3.2.1 Synchronization + Cache Coherency

As we have mentioned before, a cache invalidation protocol is not enough to synchronize the access to shared data. We still need some synchronization primitives like semaphores to synchronize the accesses. In Li's scheme, the synchronization of shared variables accesses and the cache coherent protocol is independent. To implement the readers and writer problem, he would require a read lock and a write lock. A simple implementation would be to keep both locks at a server site and a read/write lock operation would involve sending message to this server. When a reply is received from the server, the process can access the shared segment. When the access is finished, the process must send a message to the server to unlock the segment. A writer process under this scheme would be coded as :

```
Loop
   Produce something        ;
   Wait (Empty)             ;
   WriteLock (Buffer)       ;
   Deposit stuff into buffer ;
   Unlock (Buffer)          ;
   Signal (full)            ;
End
```

The reader is coded similarly. Notice that since the lock operation is separated from the access of the buffer segment, the segment is not paged in until the deposit statement is to be executed. This involves at least two messages and several invalidation messages. Hence a write cycle would involve at least five messages plus all the invalidation messages.

The Clouds researchers made two observations. First, the page in operation at the deposit statement could have been avoided if we were to page in the segment when the lock is granted. However, this would essentially mean that the lock must be associated with the segment. Second, if we force a reader to discard the segment when a read lock is released, then no invalidation message is needed when a write lock is granted. Based on these two observations, the Clouds researchers suggested the synchronization mechanism to be integrated with the cache coherency protocol. In essence, they had implemented a read/write lock for each segment at its owning partition. A request for a read lock from a process is granted when there is no writer and a write lock request from a process is granted only when there is no reader. When a lock is granted, the corresponding segment is sent to the requesting process. When a lock is released the segment is discarded. The cost for both read and write is then three messages.

3.2.2 The Cache Coherence Protocol

A segment can be freely sharable e.g, code or constant, or sharable subjected to agreements e.g, cache coherence rules. Whenever there is a segment fault, the request is always directed to the owner partition of the segment. When a partition accepts a request for a freely sharable segment, it sends a copy of it to the faulting node. However, if the segment is modifiable, then it follows the coherence protocol described next.

A DSM segment can be in one of four modes, namely none, weak read, read and read-write. The none mode guarantees exclusive access to the segment but the segment can be taken away at any time. The weak read mode provides non exclusive access to the segment but there is no guarantee on whether the segment will change during the read. The read mode provides non exclusive access to a segment and guarantees that the segment will remain unchange until the reading process has explicitly unlocked the segment. The write mode provides exclusive access to the segment and guarantees that the segment would not be taken away until the writing process has explicitly released the lock.

Satisfying a weak read is easy; the owner partition simply sends the requesting node a copy of the segment. It does not matter whether there is any other process writing the segment or not. The none mode is similar to the write mode of Li's scheme, mutual exclusion is guaranteed but the segment could be taken away at any time. Hence, synchronization of shared data access must be separately implemented. Finally, the read mode and the read-write mode is simply the combination of the locking mechanism and the coherence protocol.

When a segment is in the none mode and that there is an incoming request, the host having the segment is informed. It must release the segment and forward it to the requesting node. The owner partition must also update its information about this. Because of the interesting properties of the none mode, if a none mode request in the segment queue is followed by a read mode or read-write mode, the none mode request is discarded. This is because the segment would be immediately taken away to another node anyway.

The owner partition of a segment implements a read/write lock for the segment. When the segment is granted to other nodes for read, the owner partition keeps a set of processes that has a read copy of the segment. Incoming read requests may be granted subjected to certain fairness criteria. A process with the segment in read mode must unlock the read lock when it finishes the read. The unlock operation discards the copy and inform the owner partition of the segment. When all the processes in the read set have released the segment, the owner partition may honor any write mode or none mode request in the queue.

When a segment is in write mode , any incoming request is blocked until the writer has explicitly unlocked and discarded the segment. The discard operation will bring the segments back to the partition. This is necessary since the copy at the owner's site is no longer valid.

3.3 Contributions Of Ra

Ra is the first object oriented project to use distributed shared memory as a supporting vehicle for object relocation. Its main contribution is in recognizing that the combination of memory coherency protocol and process synchronization can improve the performance of distributed shared memory. However, in making the above statment, we must also note that the application of distributed shared memory in IVY's case and in Clouds' case are different. In IVY's case, DSM is used for parallel processing and in Clouds case DSM is used for supporting object invocation. Integrating the cache coherence protocol and synchronization protocol was easier in Cloud's case because the lock can always be acquired when the object is invoked and released when the object invocation is finished.

The cache coherence protocol of Clouds requires that a segment must be discarded when it is unlocked. This allows us to avoid the need of invalidation altogether when a segment is accessed for write. However, it does not allow us to keep the lock and the segment after an invocation and so when the object is reinvoked, the lock has to be reacquired and the segments must be refetched. A better way [Tam&Hsu 90] is for the host to keep the locks for future use but to relinquish them only when requested. The partition will now only track the last write lock. Lock requests are forwarded to this host and satisfied when the host release the lock. The host serves as a holder of the read lock set. If the request is a read request then the relock set is updated. On future write requests, a message is sent to each of these hosts to request releases on their read locks. The write lock is acquired when all the hosts with read locks have acknowledged their releases. Of course, this scheme is slower in response since we need to forward every request and to wait for invalidations if the request is a write.

4 MemNet - Hardware Implementation of DSM

Unlike the efforts described above, MemNet[Delp 88] did not start out with exploring distributed shared memory as a programming paradigm. Instead, it started with the observation that current computer communications is always done by treating the network as an I/O device. Sending a message to another node involves a system call to the kernel which invokes the appropriate protocol routines (e.g., TCP/IP) to prepare the packet. When the packet is prepared, the kernel would hand the packet to the network driver routines which handles the peculiar characteristics of the network. It was observed that the whole process takes up too much time, for copying, assembly, and disassembly. In typical implementations of layered protocols, such as TCP/IP, no more than 10-20 percent of the raw channel bandwidth is available for IPC. A way is needed so that access to the network can be more *direct.*

The MemNet [Farber 88] paradigm suggested a solution to the above problem. In the MemNet environment, all the hosts share a common address space. The address space is paged and pages are allowed to move within the system on demand. Address references to this address space are directed to an interface device within the host which acts as an intelligent memory module. This device is able to cache part of the shared memory and interact with other such devices to page in additional pages. In fact we have just shown a way to achieve remote interprocess communication without involving the kernel software. MemNet created a **short cut** from the user process to the kernel.

4.1 The MemNet Architecture

MemNet[Delp 88] is a hardware version of distributed shared memory. The shared address space is a part of the physical address space seen by each processor.

The MemNet prototype was implemented by interconnecting nodes through a 200Mbps insertion modification token ring. The custom LAN consisted of 20 parallel bit-serial lines operating at 10 Mhz, giving a gross aggregate data rate of 200 megabits per second. When the four bits of control information are subtracted, the data rate is 160 megabits per second. Each processor was connected to the system via an interface called the MemNet device.

The shared memory is structured in units of 32 byte chunks. The chunks are physically distributed across the MemNet devices, hence giving us a distributed shared memory system. A MemNet device is attached to a host processor through the processor backplane. When a reference to the shared portion of the hardware address space is passed to the MemNet device, it decides if reference can be satisfied locally. If the memory reference requires the cooperation of remote processors, then an appropriate message is sent. During this, the processor is blocked at the bus. It is *not* aware of the distributed nature of the shared memory. When a MemNet request is sent, it is circulated around the network and inspected by each MemNet device. When a MemNet device must act on a request, the response is sent by modifying the *same* request. Hence the delay of satisfying a MemNet request is predictable and minimal. Figure 4 shows the architecture of the MemNet system.

Inside the MemNet device are the interfaces to the host's system bus and the network. It also contains a large piece of memory, divided into a large cache and a reserved area. The cache is used to cache the chunks whose reserved area is a remote host. Since MemNet does not implement disk paging, a reserved area for each piece of memory in the shared address space is necessary, as we shall see.

We have briefly examined the architecture of MemNet. In the next section, we shall look at the coherence protocol.

4.2 The Cache Coherency Protocol of MemNet

MemNet uses the same cache coherence semantics as IVY, namely that a read operation must always return the most recent value of the data. There is a chunk table in each MemNet device, the table contains an entry for each chunk in the entire shared address space. Each entry consists the following status flags :

1. **valid** whether there is a valid copy of the chunk in the cache.

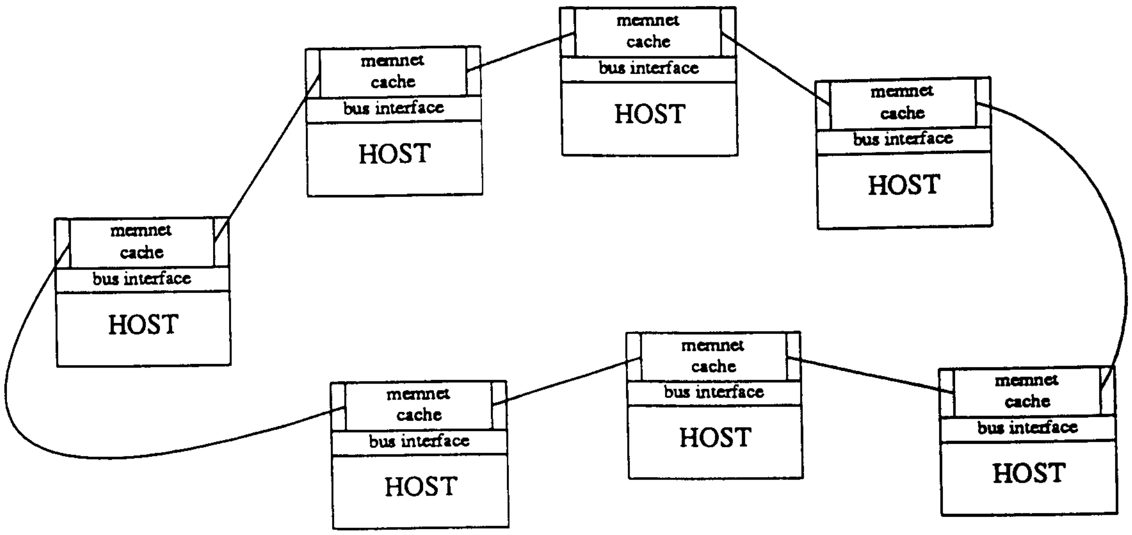

Figure 4: Architecture of MemNet

2. **exclusive** whether the host has the exclusive access right to the chunk.
3. **reserved** whether the chunk's reserved space is within the host.
4. **location** the address of the chunk copy if it is in the host.

When the device receives a read request from the processor, it will first check whether it has a valid copy of the chunk covering that address. If it does, then the request is trivially satisfied, otherwise a data request message is sent with a filler. A MemNet request is inspected by every host in turn. The read request will be satisfied by the first host that has a valid copy. The chunk is put the into the filler following the request. The request is neglected by the rest of the hosts and the chunk is finally picked up by the faulting host. The process requesting the piece of memory in the chunk is now resumed.

When the MemNet device receives a write request to an address in the shared space, it will first check whether it has a valid copy and the exclusive access to the chunk. If this is the case, then the request is trivially satisfied. On the other hand, if the device has a valid copy of the chunk but not the exclusive access to it, an invalidation request is sent. Finally, if the device does not have a valid copy of the chunk, an exclusive data request is sent. When a device receives an invalidation request, it will invalidate the chunk if the chunk is cached. The exclusive data request has a similar effect but in addition the first device that has a valid copy of the chunk must also supply the chunk before the invalidation. The blocked process will resume when the original request returns.

So far, we have not addressed what happens if cache space is full. To obtain more space for an incoming chunk, some of the chunks in the cache must be replaced. In MemNet, this problem is addressed by having a reserved area for each chunk. MemNet chose a random replacement strategy. When a device wanted to flush out a chunk from its cache space, an update chunk request is sent together with the chunk. The request is serviced by the device with reserved space for the chunk and the reserved space is updated.

4.2.1 Comparison with the IVY Scheme

MemNet used broadcasting to locate the chunks and a snooping cache mechanism to maintain cache coherency. Although MemNet used an insertion modification token ring, it used almost the same cache coherence protocol as the one used by shared memory multiprocessors with a single bus.

In a shared bus environment, read and write requests are all serialized by the bus. We have a broadcast environment where a broadcast will arrive at each node in the same order. During the write request, the invalidation and the grant of write access are done without any intervening interrupt. Since the requests are naturally serialized by the communication media and that the whole memory request process is atomic, there is no queueing in the hosts. Hence, a write can be done by simply broadcasting the invalidation and waiting for the acknowledgement. In a single token ring environment, provided that a memory request on such a ring is satisfied "on the fly", then it is not difficult to see that the ring will also have all the above properties of a shared bus. Finally, an insertion modification token ring can be viewed as a single token ring with a number of cycles "pipelined" together. As far as the hosts on the ring are concerned, they will still see the same ordering of the events in both case. Hence it is now not difficult to convince ourselves that all one needs in MemNet is a shared bus protocol.

4.3 Contributions of MemNet

MemNet provides us with invaluable experience on various aspects of the system, such as the chunk size and experience in designing hardware support for DSM. More important, it showed that by viewing remote data as residing in a high latency shared memory, one can design a system which can shorten the path of communication software. Thus, the experiment was very successful. Since the IPC time was almost a thousand times faster than a software implementation on the same processor/network architecture, and the worst case response time is bounded, Delp decided to block the processor rather than the faulting process during a chunk fault.

Further work has been done on MemNet[Sur 90]. In particular, a comparison of MemNet IPC performance and IPC performance of more traditional distributed systems such as the V kernel has been made. They conclude that the shared memory paradigm is indeed a good approach to circumventing much of the software overhead. It was also shown that the use of a fast cache rather than conventional memory does **not** result in a significant improvement in performance. This is due to the large delays in network access compared to those of main memory. The gain from using fast memory is easily offset by a miss in shared memory. On the other hand, increasing the size of the cache gives a better payoff since the hit rate is improved. However, beyond a certain limit there is no extra gain by further expanding the cache. This could be attributed to the existence of reference locality. Moreover, invalidations from the other processors also means that the hit rate is not simply controlled by both cache size and locality. Also it was shown that the performance of MemNet suffers from more than linear degradation as the number of hosts increases. We suspect the non-linear behavior could be due to the increase of network traffic, but this should be verified experimentally.

5 Comparing The Three Schemes

In this section we shall compare the three examples with the taxonomy of section 1. This includes the structure of the address space, the page location method, the method for performing invalidation, the coherency protocol, the choice of block size and also the replacement scheme. The aim of IVY is to support parallel processing and to prove that shared virtual memory implemented in software is feasible. The aim of Clouds is to use distributed shared memory as a vehicle to support mobile objects. Finally the aim of MemNet is to prove that the shared memory paradigm is a desirable way to short cut the access to network functions in the host.

The address space of IVY is a simple flat address space with no protection. This is appropriate since IVY's major concern is parallel processing where the shared address space is used to support sharing of variables between processes of the same computation. With Clouds, since the aim of having distributed shared memory is to support object mobility, and objects are made of collections of segments, the shared memory space is structured as a store of segments. That environment called for modularity, protection and also sharing between independent processes. Hence segmentation is an obvious choice[Denning 70]. MemNet focused on hardware implementation of DSM and connects the device to the processor through the address bus. Hence it was natural for it to share the physical address space.

The page location problem was thoroughly investigated in IVY. It seems that the most simple and efficient scheme is fixed distributed. The forwarding scheme has the advantage of fully distributed control. In case of Clouds, since a writer must always discard its segment back to the owner, the problem of page location does not exist. When a segment is granted in none mode, Clouds uses the equivalence of the fixed distributed scheme. Finally since MemNet is implemented on a token ring, it simply locates the most recent copy by making a broadcast.

IVY keeps a copy set for each page to avoid broadcasting invalidations. Clouds combines synchronization of access with the coherency protocol , it requires everybody to discard the segment when it has finished with it. The problem of invalidation simply does not exist in Clouds. MemNet took advantage of its token ring and used broadcasting.

Both IVY and MemNet use a single writer and multiple reader protocol. A read operation always returns the most recent version of a page. The locking protocol of Clouds provides us with similar semantics. More interesting is the weak read semantics, where Clouds returns a value copied from the owner whenever the read is executed. There is no atomicity guarantee during the read. Weak read provides us with more concurrency in the expense of tight data consistency. Application characteristics must be explored to use weak read.

The unit of synchronization in IVY is a page. Given that it is infeasible for IVY to explore fine grain parallelism anyway, a page size of 1 kbyte seems reasonable. On the other hand, Clouds' DSM supports segmentation and hence support variable object granularity. In practice, the implementation required aid from the MMU of the machines, so page sizes of the machines pose a lower bound on the segment size. But since the unit of computation is object invocation, a large block size should have no negative effect. Finally, MemNet's high speed token ring and hardware implementation allows it to explore a finer grain of parallelism. Hence, its unit of synchronization has the smallest size (32 bytes).

Page replacement was also thoroughly explored in IVY. Both replacement to disk and to other node's memories were investigated. The latter is interesting because the future workstation is likely to have a large memory. While Clouds did not address the problem explicitly, it can be presumed that replacement is done by using disk as the backing store. MemNet is unusual in that it reserves main memory to back up chunks. This is important for MemNet since it relies on a predictable and short fault repair time to avoid context switches. However, relying on this may affect MemNet's scalability for larger network latencies and address spaces.

In general, IVY tends to give every problem some considerations if not thorough. Its solutions are entirely software based. Clouds is specialized since it combines the shared data synchronization with its cache coherency protocol, thus waiving many issues. However it requires applications to explicitly lock and unlock every segment. MemNet took great advantage of its insertion modification token ring and practically addressed many of the issues simply by broadcasting. Reliance on the subnet features may be problematic when extending the MemNet approach.

6 Related Work

There is wide interest in distributed shared memory. There has been a variety of related research. Some of this research is implementation oriented (i.e., prototypes to understand performance and applications behavior) while other research focuses on DSM at the level of protocols and algorithms. The earliest work we are aware of is that of Libes [Libes 85] who implemented shared variables with function calls using TCP/IP transport. However, such access strays a bit from the transparent access we desire from DSM. Minnich and Farber [Minnich 90] have described Mether, which in an earlier incarnation [Minnich 89] was essentially a software implementation of MemNet. Mether tries to improve concurrency by relaxing the cache coherency constraints. Minnich observed that many distributed applications do not require a completely coherent image of the shared memory. Thus by providing a set of basic control primitives, it allows the programmer to decide on a policy and to exert run time control of the degree of coherency desired. Munin[Bennett 90] uses object type information to specialize the consistency control. Smith's [Smith 90a] UPWARDS system uses prefetching of DSM pages to reduce latency in high-speed wide-area networks, but since it proposes a DSM implementation strategy rather than a DSM design, we will not discuss it in this survey. The Amber System[Chase 1989] resembles Clouds in that it provides consistency semantics on objects rather than bytes.

Other papers[Stumm 90][Kessler 89] have compared different approaches to DSMs. In [Stumm 90] different ways to implement a distributed shared memory are compared. Some of them resemble the traditional approaches to distributed database, e.g., the centralized approach where fetch and store are implemented as remote operations, complete replication with central sequencers, and the optimistic approach that was suggested. These algorithms appear suited to environments where databases are the major application of DSM.

[Kessler 89] looks into an interesting problem which arises not only in the DSM setting but also in other asynchronous shared memory machines, that is, a double faulting pair. It arises when a host has read rights to a page and wants to write to it. Rather than simply asking the owner for the right to write, it must also ask for a new copy of the page, because although the faulting host has a read copy at the time of write fault, another host can also generate a write fault and could be granted before this one. When the write fault of this host is finally granted, the read copy that it has is no longer valid. This is a subtle concurrency control problem.

The problem can be solved by introducing a page version number, incremented on every ownership change. When a host with a read copy generates a write request it includes the version number of that copy. This version number is compared with that of the owner when the write access is granted. A page is transferred only if there is a mismatch. Kessler also performed a set of simulations on various coherence algorithms, indicating that this algorithm and the dynamic scheme achieves the best performance.

The remaining subsections address a variety of systems that have been designed or implemented.

6.1 The Mach Implementation

One key concept of Mach[Bisiani 88][Young 87] is the external pager. An external pager manages the backing store of its objects. Objects are accessed by mapping them into the addressable virtual memory of a process. The mapping takes place when a process invokes a system call to register the mapping. The kernel serves the call by contacting the pager of the object through some location scheme. The pager is then given a kernel port where it replies to kernel requests. After this "setup phase" is over, the memory acts as a cache of the object. The Mach kernel services a page fault by requesting the page from the object and then proceeding asynchronously. The pager satisfies the call by sending the page back to the kernel through the kernel port. A pager can also ask the kernel to flush pages and to reduce access rights by simply sending messages to the kernel port. By using these features, it is not difficult to construct a distributed shared memory. The shared memory is declared as a shared object. The pager maintains the cache coherency by keeping track of the page copies and sending them the appropriate messages, just like IVY.

6.2 Scalable Multiprocessor

Li has applied DSM to a hypercube[Li 89], a messaged passing based multiprocessor; Scheurich and Dubois [Scheurich 88] and Poplawski and Rich have also reported such work [Poplawski 87]. The performance of DSM on Li's hypercube implementation, with such an interconnection scheme, is much better than the networked computers, e.g., a page fault requires about 4 millisec. The major contribution of this effort is its potential in overcoming the limitation on the number of processors in many of the shared memory machines. The interconnection of Hypercube makes it scalable, thus implementing DSM makes a hypercube a scalable shared memory machine.

6.3 Distributed Shared Memory Accommodating Heterogeneity

Another direction [Zhou 89] is a DSM accommodating heterogeneity. This is a difficult problem, because at the page level, byte and words are the primitives, not typed data objects. The adopted approach was to tag each page with a type, thus only one type of data is allowed in a page. This is certainly a severe limitation but is still adequate for most of the array computations. Programs are also precompiled into object code for several machines to allow process migration between different computers.

6.4 Database Application

Voyager [Hsu 89] is research concentrated on the application of DSM to database problems. In Voyager, the whole database is mapped onto the virtual shared memory. Two phase locking is used as a concurrency control protocol. A transaction is done by fetching in the required data. Since a piece of data has to be read or write locked before it can accessed, Voyager also used a cache coherence scheme similar to that of Clouds. However, rather than discarding the segment during the unlock operation, it allowed the host to retain the segment. This allows locality of data references between transaction to be exploited. Since data is fetched to the node where the transaction takes place, it is unnecessary to use two phase commitment.

6.5 Fault Tolerance

Wu and Fuchs[Wu 89] investigated the recoverability issue of DSM. Although recovery is a well researched issue in database research, direct application of checkpointing requires storing multiple versions of shared pages and recording of all interprocessor communications. Moreover, cascade rollback could happen during a recovery. Their approach is to checkpoint the state of the process together with all the dirty pages that it has whenever one of its dirty pages is read by another host. This ensures that if the process fails, it will be restarted without having to rollback the one in the host that is doing the dirty read. The solution is undoubtedly a sound one, however checkpointing a process whenever there is a dirty read from a remote host could be expensive. Selectively checkpointing process state using page maps might make this technique more attractive; see, for example, Theimer's [Theimer 85] process migration scheme which uses a similar technique.

[Tam&Hsu 90] describes an interesting alternative. The page tables and the location tables are treated as a database. Since read and write requests invariably require the modification of these tables, the requests are modeled as distributed transactions. For example, a read requires modification of the tables at the owner and locally. These transactions can all be divided into subunits by sites. A transaction is considered committed as long as the initial unit is committed. Thus there is no need to perform two phase commit, as commitment of the initial subunit will cause eventual commitment of the remaining subunits. Having database-like properties means that these tables are checkpointed periodically for reliability, and that logs are written before changes are made. Hence when a host crashes, it can recover by reconstructing its page tables from these logs and the most recent checkpoint. Page requests made during the failure are brought to completion by the site initiating the first subunit. The scheme provides an elegant solution to the problem of unfinished requests caused by node failures. The cost of frequent disk accesses adds little overhead since the shared pages themselves must be logged anyway (Voyager is database application of DSM).

6.6 CapNet - A Wide-Area DSM

The research mentioned above is targeted on the local area domain. This is understandable as sharing memory across a wide area network seems unrealistic at first examination due to latency. However, if information is to be shared across the country[2], then the

[2]There is a strong analogy to the national highway infrastructure.

problem of latency is always present, independent of what IPC scheme we use. Thus, it is not at all unreasonable to investigate the possibility of a wide area distributed shared memory, in spite of the latency.

CapNet[Tam 90] is an ongoing research project at the University of Pennsylvania that follows the above argument. One of its goals is to investigate the type of network support needed to reduce latency. The researchers take a very aggressive approach and suggest that by distributing the page table in the network switches, one may reduce the number of messages to fetch a page to two. This is the minimal that one can ever achieve unless anticipation is used. In fact many of the DSM discussed in this paper either take more than this or assume a broadcast based network. When the ownership of a page is transferred, a control message is also sent to change the page tables in the switches so that the page can always be located during subsequent accesses. The approach is novel in that it suggests a way where networks can be constructed to support distributed systems directly. A similar approach has been pursued independently in the context of multiprocessing systems by [Mizrachi 89].

7 Concluding Remarks

The computation model of distributed shared memory is to make the data more accessible by moving it around. The computation model of RPC is to move operations to the location of data. There are pros and cons in both models.

RPC does not allow one to take advantage of locality. Every operation to a piece of remote data induces communications. Operations on data must be predefined. However, this also provides us with a very good handle on addressing the heterogeneity problem. Distributed shared memory allows us to take advantage of locality by moving data to the local node. It also allow us to do caching so that the response time is improved. Mobility implies keeping track of the location of the data and caching implies that certain notions of consistency must exist between the copies. When a piece of data is on its way to a host, it cannot be processed. This could imply that the RPC model may be better if the data is modified frequently.

Different applications have different requirements on data consistency. Some of them have tight requirements, e.g., parallel processing, while others have a looser requirements, e.g., name space management. It is non trivial to capture all kinds of requirements at the system level. Most of our examples follow the same consistency semantics i.e, a read would always return the most recent value. Sharing and caching is desirable but it might be better and more efficiently implemented at the application level, hence the notion of problem oriented shared memory advocated by Cheriton [Cheriton 86].

Protection is always a concern when address spaces are shared. Both message passing and RPC provide an effective firewall between processes. Capability has been suggested as a protection mechanism for DSM, however the overhead induced may mean that architectural support is important. Smith [Smith 90b] has suggested using cryptography as a scheme to provide traditional memory protection semantics in a DSM.

Message passing requires a programmer to handle communications explicitly and does not support data sharing directly. However, if processes communicate because they want to synchronize or to share well defined information occasionally, message passing seems to be more natural. Moreover, since the peer process and the act of communication is visible at the programming level, message passing also provide us with better handles on handling process failure. On the other hand, it is unclear how a failed page fault is handled.

Obviously neither message passing nor shared memory could be overwhelmingly better than the others. Distributed shared memory is far from mature. However, it is certainly valuable to have it as part of the operating mode of our distributed system. Its full use would only be demonstrated when programmers have it as an alternative and start exploring it for applications.

References

[Agarwal 88] Anant Agarwal, Richard Simoni, John Hennessy, Mark Horowitz. "An Evaluation of Directory Schemes for Cache Coherence", *Proceedings, 15th International Symposium on Computer Architecture*, June 1988, pp. 280-289

[Arch 86] J. Archibald and J. Baer. 'An Evaluation of Cache Coherence Solutions in Shared-Bus Multiprocessor'. *ACM Transactions on Computer Systems*, February 1986.

[Auban 88] J. B. Auban, P. Hutto, Y. Khalidi. 'The Architecture of the Ra Kernel' *Technical Report GIT-ICS-87/35, Georgia Institute of Technology, Computer Science* ,1988

[Bennett 90] John K. Bennett, John B. Carter, and Willy Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence", *Proceedings, 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990, pp. 168-175

[Bensoussan 72] A. Bensoussan, C.T. Clingen, and R.C. Daley, "The MULTICS Virtual Memory: Concepts and Design", *Communications of the ACM* , May 1972

[Birrell 86] A.D. Birrell and B.J. Nelson. 'Implementing Remote Procedure Call'. *ACM Transactions on Computer Systems*, 1984.

[Bisiani 88] Roberto Bisiani and Alessandro Forin "Multilingual Parallel Programming of Heterogeneous Machines" *IEEE Transactions on Computers*, August 1988, pp. 930-945

[Chase 89] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield, "The Amber System: Parallel Programming on a Network of Multiprocessors", *Technical Report 89-04-01, University of Washington, Department of Computer Science and Engineering*, September 1989

[Cheriton 86] D. Cheriton. 'Problem-Oriented Shared Memory : A Decentralized Approach to Distributed System Design" *IEEE, Sixth International Conference on Distributed Computing Systems, 1986.*

[Daley 68] R.C. Daley and J.D. Dennis. 'Virtual Memory, Processes, and Sharing in Multics'. *Communications of the ACM, 11(5) : 306 - 312*, May 1968.

[Das 88] P. Dasgupta, R. LeBlanc, W. Appelbe. "The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work", *International Conference on Distributed Computing System IEEE*, 1988.

[Delp 86] G. Delp, D. Farber. 'MemNet : An Experiment on High-Speed Memory Mapped Network Interface'. *Technical Report, 85-11-IR University of Delaware, Computer Science Department*, 1986.

[Delp 88] G. Delp. 'The Architecture and Implementation of MemNet : A High Speed-Shared Memory Computer Communication Network" *Ph.D Thesis, University of Delaware, Computer Science Department*, 1988.

[Denning 70] Peter J. Denning. "Virtual Memory". *ACM Computer Survey*, September 1970.

[Farber 88] D. J. Farber. "Some Thoughts on the Impact of High Speed Network on Processors", *DSL notes, Department of Computer and Information Science, University of Pennsylvania*, 1988.

[Fowler 86] R. J. Fowler. 'Decentralized Object Finding Using Forwarding Addresses'. *Ph.D thesis, University of Washington, Department of Computer Science and Engineering,* , 1986.

[Hsu 89] M. Hsu, V. Tam, 'Transaction Synchronization in Distributed Shared Memory' *Technical Report TR-05-89, Harvard University, Department of Computer Science* 1989.

[Kessler 89] R. E. Kessler and Miron Livny, "An Analysis of Distributed Shared Memory Algorithms", *Proceedings, 9th International Conference on Distributed Computing Systems* , 1989, pp. 498-505

[Li 86] Kai Li. 'Shared Virtual Memory on Loosely Coupled Multiprocessors'. *Ph.D Thesis, Yale University, Department of Computer Science*, 1986.

[Li 89] Kai Li and R. Schaefer, "A Hypercube Shared Virtual Memory System", *Proceedings, International Conference on Parallel Processing*, 1989, Volume I, pp. 125-132

[Libes 85] Don Libes, "User-Level Shared Variables", *Proceedings, Tenth USENIX Conference*, Summer 1985

[Minnich 89] Ronald G. Minnich and David J. Farber, "The Mether System: A Distributed Shared Memory for SunOS 4.0", *Proceedings, Summer 1989 USENIX Conference*

[Minnich 90] Ronald G. Minnich and David J. Farber, "Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory" *Proceedings, 10th International Conference on Distributed Computing Systems* , Paris, France, June 1990

[Mizrachi 89] H. E. Mizrachi, J. L. Baer, E. D. Lazowska, and J. Zahorjan, "Extending the Memory Hierarchy into Multiprocessor Interconnection Networks: A Performance Analysis", *Proceedings, International Conference on Parallel Processing*, 1989, Volume I, pp. 41-50

[Nel 84] D. L. Nelson and P.J. Leach "The Architecture and Applications of the Apollo Domain", *IEEE Computer Graphics*, April 1984, pp. 58-66

[Peterson 86] J. Peterson, A. Silberschatz. "Operating Systems Concepts", *Addison-Wesley Publishing Company*, 1986.

[Poplawski 87] D.A. Poplawski and D.O. Rich, "Code Paging on Hypercubes", *International Conference on Parallel Processing* , August 17-21, 1987

[Powell 83] Michael L. Powell and Barton P. Miller, "Process Migration in DEMOS/MP" *Ninth ACM Symposium on Operating Systems Principles* , 1983

[Ram 88] U. Ramachandran, Y. Khalidi. 'An Implementation of Distributed Shared Memory'. *Technical Report GIT-ICS-88/50*,December, 1988

[Reed 79] David P. Reed, R. Kanodia. "Synchronization with Eventcounts and Sequencers". *Communications of the ACM* , February, 1979.

[Scheurich 88] C. Scheurich and M. Dubois, "Dynamic Page Migration in Multiprocessors with Distributed Global Memory" *8th International Conference on Distributed Computing Systems* , June 1988, pp. 162-169

[Smith 88] Jonathan M. Smith, "A Survey of Process Migration Mechanisms", *ACM Operating Systems Review* , July 1988, pp. 28-40

[Smith 90a] Jonathan M. Smith, "Anticipation in Very High Speed Networks", Distributed Systems Laboratory Technical Report, Department of Computer and Information Science, University of Pennsylvania, 1990 *(submitted for publication)*

[Smith 90b] Jonathan M. Smith, "Security of Distributed Virtual Memory", Distributed Systems Laboratory Technical Report, Department of Computer and Information Science, University of Pennsylvania, 1990 *(submitted for publication)*

[Spector 82] Alfred Z. Spector, "Performing Remote Operations Efficiently on a Local Area Network", *Communications of the ACM*, April 1982.

[Stumm 90] Michael Stumm, Songnian Zhou. "Algorithms Implementing Distributed Shared Memory", *IEEE Computer*, May 1990.

[Sur 90] S. Sureshchandran, Timothy A. Gonsalves. "Performance of the MemNet Distributed Shared Memory Architectures", *TR-CSE-90-02 Department of Computer Science, Indian Institute of Technology*, January 1990.

[Tam 90] Ivan Ming-Chit Tam and David J. Farber "CapNet - An Alternative Approach to Ultra High Speed Networks", *Proceedings, International Communication Conference* , 1990

[Tam&Hsu 90] a-On Tam, M. Hsu "Fast Recovery in Distributed Shared Virtual Memory Systems", *Tenth IEEE. International Conference on Distributed Computing Systems*, May 1990

[Tanenbaum 85] Andrew S. Tanenbaum and Robbert Van Renesse, "Distributed Operating Systems", *ACM Computing Surveys* , December 1985, pp. 419-470

[Theimer 85] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton, "Preemptable Remote Execution Facilities for the V-System", *Proceedings, 10th ACM SOSP* , 1985, pp. 2-12

[Wu 89] K. L. Wu, W. K. Fuchs. "Recoverable Distributed Shared Virtual Memory: Memory Coherence and Storage Structures". *The Nineteenth International Symposium On Fault-Tolerant Computing, IEEE..*, 1989

[Young 87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black and R. Baron "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System" *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles* , November 1987, pp. 63-76

[Zhou 90] S. Zhou, M. Stumm, T. McInerney, "Extending Distributed Shared Memory to Heterogenous Environments", "Proc. 10th Int'l Conf. Distributed Computing Systems, 1990.