Access Consistency Memory Model for Messengers[1]

Murhimanya Muhugusa Giovanna Di Marzo
Christian Tschudin[2]
Jürgen Harms
Centre Universitaire d'Informatique, University of Geneva
24, rue Général Dufour, CH-1211 Genève 4
Phone: +41 22 705 76 43, Fax: +41 22 705 77 80
e-mail: muhugusa@cui.unige.ch

Cahier du CUI n^o 107

November 15, 1996

[1]This work is supported by the Swiss National Fund for Scientific Research (FNSRS) grant 20-40631.94

[2]Institute fur Informatik, University of Zurich

Abstract

The messenger paradigm is an alternative to computer communication based on the exchange of programs called messengers which are afterwards executed, instead of messages that are interpreted. The communication by messenger paradigm can be used to implement both low level software such as communication protocols and high level software such as distributed applications. Using the messenger paradigm for computer communication requires that each host be "equipped" with a runtime environment called messenger platform that can execute messengers that reach the platform. M∅ *is one such platform. MOS is a distributed micro-kernel designed for the efficient execution of messengers and implemented on i386 machines; it contains a* M∅ *platform. It supports the concurrent execution of messengers on a given node. Messengers running on the same node can exchange data through a common store. Messenger behavior is expressed in the* M∅ *messenger language understood by all messenger platforms.* M∅ *is an interpreted language. Messengers see the memory provided by a MOS as a set of dictionaries where information is stored as pairs consisting of a key and a value. The key is used to access the value associated to a given information. In this paper, we present a memory consistency model, called access consistency, that is well suited to implement a distributed shared memory to extend the common store available for messengers running on the same MOS node to spawn a network of MOS nodes; i.e, to allow messengers to share information irrespective of their physical location in the network.*
Keywords: memory consistency model, access consistency, messenger, M∅, MOS.

Chapter 1

Introduction

Distributed shared memory (DSM) is an attractive model for information sharing between processes or threads that execute on different computers linked through a network. When a DSM system is used, each process can reference data located everywhere in the network as if it was local to the node where the process is running. The memory of the different nodes composing the DSM is seen as a cache of the shared memory. Like in a caching system, access to non local data (data not available in the local cache) results in the data being transferred from a remote node to the local cache. The transfer of data between the different caches over conventional networks is very costy. Hence, most DSM systems try to reduce the traffic between the different caches.

One way of reducing traffic between caches (nodes) is by replicating data on several nodes. Data is then accessed locally on all the nodes where it is replicated. However, the price to pay for data replication is that of maintaining the coherency of the replicated data. This means that, modifications done on data have to be applied somehow on all the replicas. As a consequence, maintaining data coherency generates activity on several nodes and also some amount of network traffic. In the worst cases, maintaining coherency of replicated data can generate more network traffic than that necessary to achieve data sharing without replication.

In fact, the amount of network traffic generated by a DSM system is highly dependent of the "protocol" (memory consistency model) used by the DSM system to maintain the coherency of replicated data. Memory consistency model is therefore an important issue in designing a DSM system and the choice of a memory consistency model is one of the critical decisions that have to be taken by any designer of a DSM system. Many memory consistency models [Mos93] have been designed for different kinds of DSM systems, but none of them is suited for all DSM systems. The choice of a memory consistency model depends on a number of parameters, among which we can mention the type of applications for which the DSM system is designed and the system (hardware and software) on top of which the DSM system has to be implemented.

In order to understand why a specific memory model is needed for mes-

sengers, we will present first briefly, the messenger paradigm and how memory is managed in the MOS micro-kernel. Access consistency is designed for implementing DSM with messengers and for messengers running on a network of MOS nodes.

1.1 The Messenger Paradigm

Messengers [Tsc93] are mobile threads of execution useful for structuring distributed algorithms [DMMTH95]. The execution of a messenger takes place in a *messenger platform*. One such platform is the MØ platform [Tsc94]. Several messenger platforms are connected by unreliable channels through which messengers are sent as simple data packets.

Inside a given platform, messengers are *sequential processes* executing *concurrently and in parallel*. They *coordinate their execution* by the means of (1) a shared *dictionary*, and (2) *messenger queues*. The dictionary is a shared data structure accessible to all messengers. A messenger can insert new data, change or remove old data in/from the dictionary. A messenger is able to insert itself in a queue, its execution is then stopped until it reaches the head of the queue. Messengers can *create at run-time* (1) new data, (2) new messengers in the platform where they are executing, and they can *move* themselves or *send other messengers to other platforms*.

To summarize, messengers are anonymous and autonomous sequential processes, executing in parallel and able to move between platforms and to coordinate their work by the means of shared data structures and messenger queues.

1.2 The MOS Distributed Micro-Kernel

MOS (Messenger based Operating System) [TDMMH94] is a distributed micro-kernel designed for the efficient execution of messengers and implemented on i386 architectures. As is the case for every micro-kernel, the main challenge for MOS is to determine the basic functionality (minimal set of services) to confine in the micro-kernel in order to get a very small and truly generic micro-kernel. The driving philosophy of MOS with this respect, is to avoid location transparency at the micro-kernel level. This is practically achieved by providing only local services at the micro-kernel level and by shifting all inter-node services, i.e., services requiring cooperation between different nodes at the operating system servers level. The main advantage of this approach is that there is no "hard-wired" protocol inside the micro-kernel.

1.2.1 The Structure of MOS

In fact, in MOS, each micro-kernel is a MØ messenger execution platform offering only local services to messengers. Each node contains such a platform; and when messenger platforms are linked through a network, they constitute a distributed micro-kernel, even though there does not exist any kind of cooperation between

them. However, as a messenger can create other messengers for execution on remote messenger platforms and because messengers can be used to implement arbitrary protocols, messengers are used in MOS to implement inter-node (inter-platform) services. Cooperation between nodes is achieved in this way at the messenger level and not at the micro-kernel level.

In MOS, the different platforms are linked through unreliable channels. Messengers use the channels to move from platform to platform. When a messenger moves, it is packed by its origin platform which then outputs it on the specified channel. If the packed messenger reaches its destination platform in whole and without corruption, it is unpacked and is unconditionally turned into a messenger thread and executed by the platform concurrently with other messengers, provided there are enough resources to do so.

Actually, three levels of abstraction exist in MOS (see figure 1.1). The first

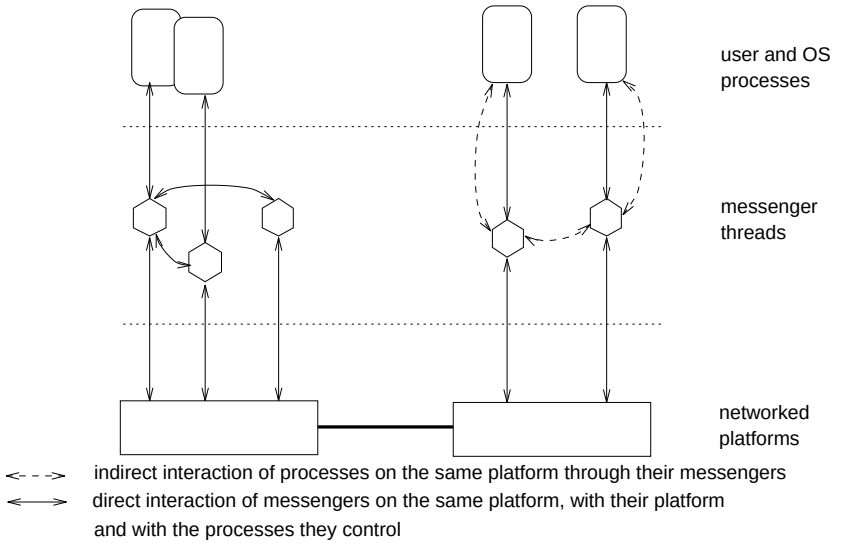

Figure 1.1: The three levels of abstraction in MOS.

level corresponds to the distributed micro-kernel described above as a set of messenger platforms linked through a network. Messenger platforms offer basic services for messengers which occupy the second level of abstraction. Messenger platforms, together with the hardware where they execute supply a virtual machine that understands the messenger language. Messengers are concurrent threads of execution written in the messenger language; they are interpreted by the virtual machine provided by messenger platforms. At the level of messengers, sits the real operating system, i.e., the necessary servers that implement OS functionality; therefore, OS servers are implemented in MOS by messengers. However, for efficiency reasons, OS servers should not be completely implemented in an interpreted language. For this reason, and for the need to execute user programs compiled in native CPU-code, MOS allows the execu-

tion of native code. Hence, the third level of abstraction corresponds to the classic abstraction of process. At this level,we have processes executing native CPU-code under the control of messengers. A messenger can control the execution of only one process, but it is possible to have a messenger without a process under its control. As is shown in figure 1.1, messengers executing on the same platform can interact directly. A messenger can interact also with the process under its control and with its platform. Two processes executing on the same platform cannot interact directly, their controlling messengers must mediate their interaction (e.g. achieve interprocess communication).

At bootstrap time, a first messenger is injected into the network of messenger platforms. This messenger will create other messengers that will finally populate all the platforms. All these messengers will collaborate to offer the services of the operating system. One can imagine in an extreme case, that different initial messengers implementing different operating systems are injected into the network of platforms. Each of them may get control of a subset of platforms; this will result in the partioning of the system into disjoint sets of nodes that run different operating systems. For example, each set of nodes can implement its "own" process and memory management policies. What is unusual is that, even in this extreme case, messengers running on nodes belonging to different sets are still able to interact. Moreover, messengers can move between nodes running different operating systems.

1.2.2 Services offered by the MOS Micro-Kernel

As mentioned above, the MOS micro-kernel presents to messengers a virtual machine offering only local basic services. The MOS micro-kernel does not contain policies, it only offers mechanisms that can be used by messengers to implement and enforce various kinds of policies. The following principles are applied to all services provided by the MOS micro-kernel:

- A key mechanism is used for controlling access to data objects in memory:a messenger cannot access an object for which it does not have the right key. Keys can be seen as capabilities. They are used to locate objects in memory and to specify the operations allowed on them. When an object is stored in memory, its creator must associate to it at leat one key. Then the object creator must publish the key to the other messengers that need to access the data. An object creator or owner can grant selectively access rights on the object to different messengers. Indeed, if the object creator associates multiple capabilities specifying different access rights to the object, and if it publishes selectively the capabilities to different messengers, these messengers will be grant different access rights on the object.

- A currency mechanism is used for uniform resource management: messengers pay for all the resources they consume for their execution; namely, CPU time, memory and network bandwidth. When a messenger is no more able to pay for the resources it consumes, it is silently removed from

the system. The price of different resources are dynamically adjusted according to the demand from messengers. By adjusting the credit given to different messengers, it is possible to enforce various kinds of priorities. Practically, each messenger has an account from which the system deduces charges resulting from the execution of the messenger. A messenger can transfer money to another messenger executing on the same platform. This is done by letting the first messenger withdraw money from its account and deposit it in the account of the other messenger. Finally, a number of messengers can share an account; this is desired when the messengers cooperate to achieve a common task. Their common account is used to pay for the resources they consume to accomplish the task. Of course, the MOS micro-kernel ensures that this currency mechanism is not forgeable; i.e., a messenger cannot increase its assets fraudulently or cannot withdraw money from another messenger's account.

Even though all the services provided by the MOS micro-kernel are highly intertwined, we group them in (a) thread management services, (b) memory management services, (c) execution of native code and (d) other services.

Thread management includes (a) the execution and the scheduling of messengers, (b) the creation of messengers, and (c) the synchronization of messengers. In the next section, we detail the memory management service in the MOS distributed micro-kernel.

1.3 Memory Management in MOS

The MOS micro-kernel provides only a basic memory management service for messengers running on the local node. Messengers request memory to the micro-kernel by creating and storing data structures into memory. No virtual memory services are provided by the MOS micro-kernel. A messenger can request that a data structure be stored in a private memory region for the messenger, or in a region of memory shared with other messengers. Also, messengers are charged for memory they consume for their execution. This done by requesting that each data structure be funded. A data structure that is no longer funded is removed from the memory. For data structures stored in the private memory of a messenger, the messenger account is used to fund the data. Shared data structures are funded differently: a messenger can attach an account to a shared data item, and a number of such accounts can be attached to a data item. These different accounts are then used to fund the data item. In this way each messenger that needs the data can decide to sponsor it independently of other messengers. Also this ensures that a data item is not removed from the memory if a misbehaved or ill-intentioned messenger attaches an empty account to the data if other non empty accounts are attached to the data.

In MOS, messengers and processes have different views of the memory inside which they are executing. On one hand, a messenger sees the memory of the platform on which it is executing as a set of three dictionaries. One of these

dictionaries is private to the messenger.[1] It is used by the messenger to store information—data and code—that is not shared with other messengers. The other two remaining dictionaries[2] are accessible to all messengers running on the same platform and are used for information exchange between messengers. Information is stored in dictionaries in pairs consisting of a key and an associated value. We will note such a pair (k, v) where k is the key and v the value associated to it. Both k and v can be arbitrary data objects[3] supported by the underlying messenger language. Normally messengers access information stored in memory (one of the three dictionaries), by proving its associated key. One operator is used to create a new pair (k, v) in memory or to change the value of v if the pair already exists in memory. A second operator allows a messenger to access the value v if it provides the right key k associated to it. And a last operator is used to remove a known pair from a dictionary.[4] One can see the *address space* of a messenger as a collection of data items—portions of memory—disseminated into three dictionaries (see figure 1.2). Indeed, even if each of the two shared dictionaries is potentially entirely accessible, the messenger has effectively access only to parts of them, namely those containing information for which the messenger knows the associated key. What is unusual here is that there is no notion of "address". Information stored in memory is accessed indirectly via its associated capabilities. In fact, one part of the information is used to access the remaining part.

On the other hand, processes have a rather traditional view of memory, i.e., a flat space of addressable memory words subdivided into three regions: a read-only region for the code, a read-write region for data and another read-write region for the heap and the stack. Contrary to the messenger case, the address space of a process is linear and completely private. A priori, two processes will have completely unrelated address spaces protected each from the other and therefore will have access to different objects in memory. Also, the address space of a messenger is protected to the process under the control of the messenger. A process cannot access objects lying in the address space of its controlling messenger. However, a messenger can access memory regions mapped in the address of the process under its control. The main reason for this, is to allow a messenger initialize the address space of its associated process with the right information such as code, data, and other state information before launching theexecution of the process.

To enforce the two views of memory described above, two distinct mechanisms are used to achieve protection. On one hand, in the messenger space (the second level of abstraction), address space protection is carried out by the messenger platform.The messenger platform detects any attempt from a messenger to access information not lying in the messenger's address space—when

[1] In M0, the private messenger dictionary is called `localdict`.

[2] Called respectively `globaldict` and `servicedict` in `MØ`.

[3] In `MØ`, k is usually a `name` or a `key` object.

[4] `MØ` provides the `define`, `get` and `undef` operators. They all take a dictionary and the object to use as the key k for the data item being defined, accessed or destroyed. The `define` operator takes a third parameter specifying the value v to associate with the key k.

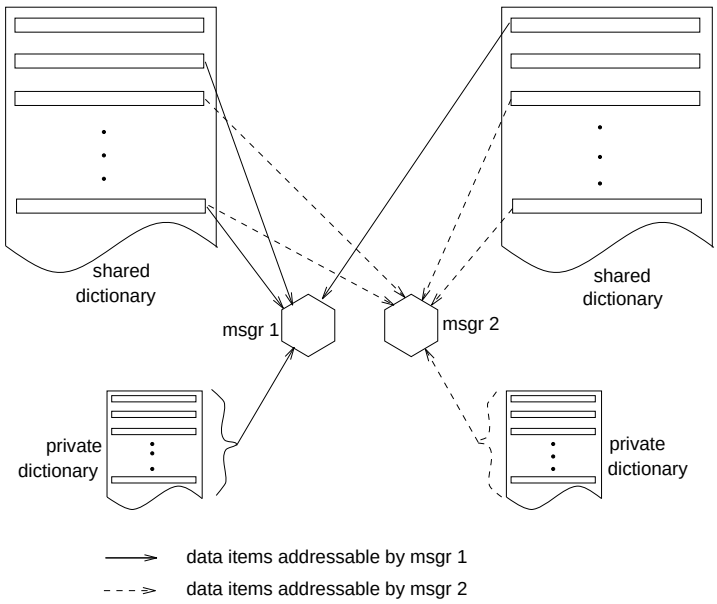

Figure 1.2: Two messengers and their "address spaces".

the messenger does not provide the right key to access the information—and generates an "error condition". If the faulting messenger is not prepared to handle such a condition, it is aborted.

On the other hand, MOS uses the facilities provided by the memory management unit (MMU) to achieve protection at the process level. However, address space protection and management at the process level is not done by the MOS micro-kernel. It is the responsibility of messengers controlling the execution of processes. The MOS micro-kernel provides the basic services to allow messengers to do the job as they need. Indeed, a messenger can create an object called a *page map* to represent the address space of the process executing under its control. Afterwards, the messenger can add objects called *pages* in the page map. The page map and the pages it contains define the portions of the address space of the process that are mapped into main memory.

A messenger can dynamically change the address space mapping of the process under its controls. Furthermore, page maps and pages can be shared between messengers as is the case for all other objects. In this way, messengers can realize sharing between processes running under their control, either at the address space level(sharing address spaces) or at the physical level (sharing physical pages). Sharing at the address space level is achieved between two processes when their address spaces are mapped identically into main memory. In this case, their address spaces contain exactly the same pages, and these pages are mapped to the same physical frames. On the contrary, sharing at the physical level is achieved when two processes share a number of physical frames. It is not

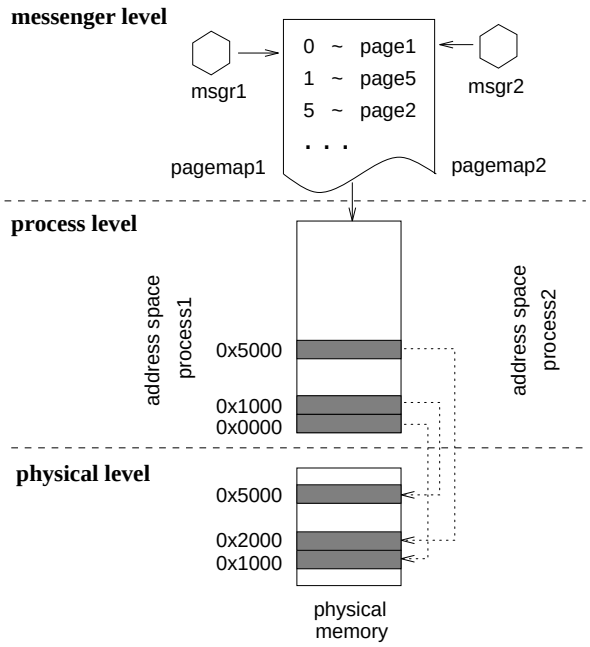

Figure 1.3: Sharing an address space.

necessary that the shared physical frames be the mapping of the same pages. For example page P_1 of one process and page P_5 of the second process can be mapped on the same physical page. In the notation used here, the subscript indicates the page number in the process address space.[5]

Figure 1.3 illustrates how messengers can realize sharing of the address spaces of the processes under their control. The two messengers share the same page map object which is used to represent the address space of both processes. Consequently, the two processes share physical pages. Correspondig addresses in address spaces of the two processes reference the same objects. This sharing scheme can be used to implement multi-threading. The two native code processes can be seen as different flows of control of the same process. Finally figure 1.4 shows how messengers achieve sharing of physical pages for processes under their control. The two processes have distinct address spaces described by two distinct page maps. For the first process, only pages 0,1 and 5 of its address space are mapped into physical memory. They are respectively mapped on physical frames 1, 5 and 2. And the second process has pages 5, 8 and 9 of its

[5]In the `M0` messenger platform, page maps are created with the `createpagemap` operator, and pages with the `createpage` operator. Page maps are handled as dictionary; the operators for handling dictionaries are used to add/remove pages in/form page maps. Only integer values can be used as keys to add pages in a page map. The `setpagemap` installs the specified page map to represent the address space of the process under the control of the calling messenger, and the `getpagemap` operator returns the page map describing the address space of the process under the control of the calling messenger.

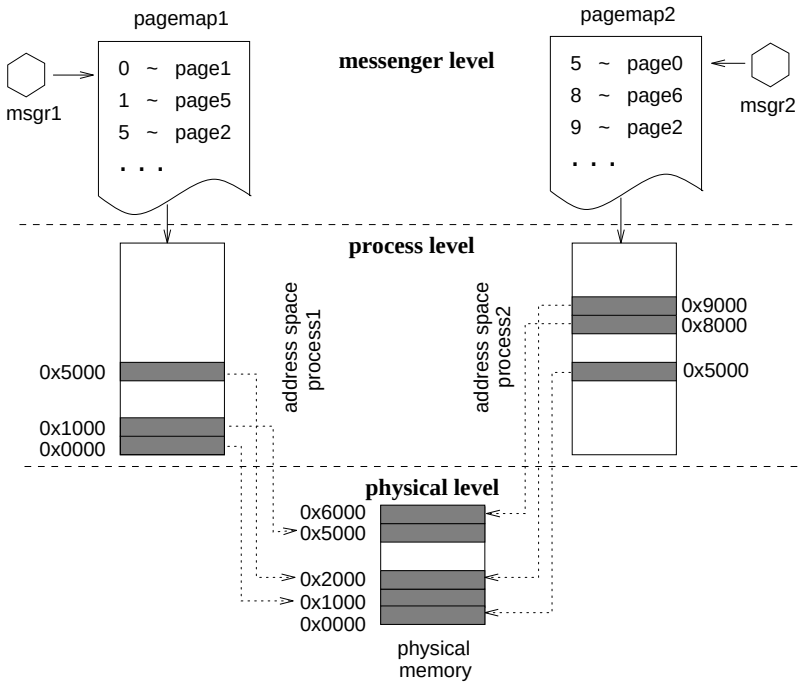

Figure 1.4: Sharing physical pages.

address space mapped respectively on physical frames 0, 6 and 2. Hence physical frame 2 is shared by the two processes. However, objects lying on physical frame 2 are not referenced by the same addresses in the two processes. In the two cases of sharing illustrated above, processes executing native code are not aware that they are sharing a number of physical pages or their whole address space. All happens transparently to the two processes. Therefore, the processes have no means to ensure consistency of the information they are sharing. More precisely, without any additional protection means, the processes cannot detect situations of racing. However, the two sharing schemes are sufficient for the sharing of read-only information for processes running on the same messenger platform.

1.4 Memory Consistency Models

In a uniprocessor system, each write access to memory is instantaneously visible to all running threads. In a multiprocessor system with distributed shared memory or in a multicomputer system, this is no longer true. There is a "somewhat long" delay between a write access to shared memory at one processor and the time this write access becomes visible to other processors. Moreover, in a uniprocessor system, all write accesses to memory are totally ordered and are

therefore seen in the same order by all threads. This is difficult and costly to achieve in a multiprocessor system. And depending on how local write accesses to shared memory are made visible to remote processors, these writes can be seen indifferent order by remote processors. For example, a write from processor P_i at location x can be seen at processor P_j before a write from processor P_k at the same location, while the write from P_k is seen before the write from P_i at processor P_l.

It is impossible for a programmer to write programs that execute correctly on a system if the memory behavior as determined by the memory consistency model[6] is not known a priori. Tanenbaum gives in [Tan95] the following definition: *A memory consistency model is essentially a contract between the software and the memory. If the software agrees to obey certain rules, the memory promises to work correctly.*

Defining a memory consistency model is essentially fixing the order under which the different processors see memory accesses, i.e., fixing the different sequences allowed to be observed by the processors. Two memory consistency models will differ in the sequences of memory accesses they allow the different processors to observe. Consistency models which impose more restrictions on the memory access sequences[7] will allow the programmer to express easier correct programs than models that impose less restrictions. On the other end, less restrictive memory models exhibit better performance than more restrictive models. Therefore designing a memory model is making a trade-off between the desired memory performance and the desired easy of programming.

memory model determines the performance of the DSM system, how easy a programmer can express a correct program ...

Different memory consistency models have been defined [Mos93, KNA93]. We present some of them in the following sections. In [Mos93], Mosberger discusses memory consistency models and their influence on software in the context of parallel machines and focuses on the influence that weakened consistency models have on language, compiler and runtime system design. Some researchers [Mis86, ABJ+93, MRZ95] are interested in formalizing memory consistency models in order to compare the various models and to devise efficient implementations.

Sequential Consistency (SC)

Contrary to strict consistency which allows only one sequence of memory accesses, and therefore does not allow indeterminism, all other consistency models allow multiple sequences of memory accesses. Lamport [Lam79] defined a SC memory to be a memory that allows as the result of the execution of a (mul-

[6] The memory behavior determines precisely the different sequences of memory accesses visible by the different processors.

[7] Memory model M_1 is said to be *more restrictive* or *stronger* that model M_2 if the set of allowed memory access sequences under M_1 is a subset of the memory access sequences allowed under M_2. If M_1 is stronger than M_2, M_2 is said to be *less restrictive* or *weaker* than M_1.

tiprocess) program, any valid interleaving of the memory operations specified by each process. More precisely, a memory is SC if it satisfies the following condition:

> *The result of any execution is the same as if the operations of all processes (processors) were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

The behavior of a SC is that observed when a multiprocess program is executed on a uniprocessor system. Therefore, SC can be seen as a model that allows a distributed shared memory system to hide the distributed nature of the shared memory.

SC is considered to be the ideal memory model that has to be provided by a distributed shared memory system.[8] For this reason, most weaker consistency models that have been defined—because implementations of SCare known to have bad performance—specify conditions that must be satisfied by a program in order to get executions that are sequentially consistent. Release consistency and entry consistency are two such weaker models. They are all based on the fact that many programs expect to see a consistent memory only at some points of their execution. As an example, write operations performed by a process in a critical section need not be seen by other processes waiting to enter that critical section until they get access on the critical section. It is sufficient that the waiting processes see these write operations only when they enter in the critical section.

Weakly consistent systems

Dubois et al. [DSB88] have observed that SC consistency is an overly restrictive memory model and proposed weak ordering as an alternative to SC. Weakly consistent systems comprise those memory models that use weak ordering without making distinction between accesses to shared memory. Weakly ordered systems, presented in the next section, also use weak ordering but make distinction between accesses to shared memory.

The aim of both weakly consistent systems and weakly order systems is to improve the performance of shared memory systems by weakening the consistency guarantees provided by the system. This is possible because programs do not need to see a consistent memory all the time in order to execute correctly. However, weakening the memory consistency increases the constraints on program behavior in order to get correct execution. This means that, the programmer has to deal with the weak guarantees provided by the memory and to be aware that programming for weakly consistent memory systems can be error prone. Most programmers are accustomed to SC, making a transition to a more restrictive memory model is not an easy task.

[8] Many implementations of parallel and concurrent programs assume a SC memory model. Programming for a weaker memory model is more difficult and therefore more error prone.

Some weakly consistent memory models are Processor Consistency [Goo89, GLL+90], PRAM Consistency [LS88] and Causal Consistency [LS88].

Weakly ordered systems

Weakly ordered memory systems make synchronization operations explicit to the memory system, and consistency maintenance is done only at synchronization points. They guarantee correct execution for programs that are properly labeled or data-race-free. If all accesses to shared memory are bracketed inside synchronization operations, the memory guarantees to provide a SC behavior. This reduces the burden imposed on programmers since they are expected to write data-race-free programs.

Many weakly ordered memory systems have been defined. We present in this section, release consistency, lazy release consistency, entry consistency and scope consistency. In the presentation below, we will use the following definition from [DSB86]: *An update to a memory location is said to perform with respect to processor p_i at a point in time when a subsequent read of that location by p_i returns the value written by the update.*

Release Consistency (RC): Both release consistency and entry consistency distinguish accesses to shared memory into *ordinary accesses* and *synchronization accesses*. A synchronization access is performed on a *synchronization variable* and can be either an *acquire* or a *release*. An acquire is performed when entering a critical section and a release is performed when leaving a critical section. A distributed shared memory is said to be release consistent if it satisfies three conditions [GLL+90, IDFL96]:

1. *Before an ordinary access to a shared variable is performed, all previous acquires done by the processor must have completed successfully.*
2. *Before a release is allowed to be performed, all previous reads and writes done by the processor must have completed.*
3. *The acquire and release accesses must be processor consistent.*

The first condition states that a processor ensures to get up-to-date values of shared variables if it performs an acquire before accessing shared variables. The second condition states that modifications done to shared variables by a processor may not be made visible to other processors before the modifying processor has performed a release access. And the last condition ensures that writes performed to shared variables by any processor are seen by all other processors in the same order, although two writes from different processors may be seen by any two processors in a different order. Hence a RC memory promises a program will get sequential executions if all accesses to shared variables are bracketed by acquire/release

pairs, i.e., all accesses to shared variables are performed inside critical sections.

Lazy release consistency (LRC): Lazy release consistency [KCZ92, Kel95] is a variation[9] of RC that delays the propagation of modifications to shared variables done inside a critical section. Instead of propagating those modifications to all other processors when the modifying processor performs a release, the modifications are made visible to any processor only when it tries to access the shared variables, i.e., when it performs an acquire. Compared to RC, LRC reduces the number of messages exchanged between processors to maintain memory consistency.

Entry Consistency (EC): Entry consistency has been defined by Bershad et al. [BZ91, BZS93]. As for RC, EC ensures that a processor has a consistent view of shared memory only when it enters a critical section. However, contrary to RC which ensures that a processor gets up-to-date values of all shared variables when it performs an acquire, EC ensures that only shared variables accessed inside the critical section are updated. This is done by associating each shared value to a synchronization variable, i.e., each critical section is guaranteed to protect only a subset of shared variables. In this way, a number of critical sections protecting disjoint sets of shared variables can be executed concurrently by different processors.

To further increase the degree of parallelism between different processors, EC distinguishes exclusive and non-exclusive accesses to shared variables. More than one processors can have non-exclusive access to the same set of shared variables at the same time. But when a processor holds to a set of shared variables in exclusive mode, no other processor is allowed to access the same set of variables at the same time either in exclusive mode or in non-exclusive mode. More formally, given that the synchronization variable (s) is used to control access to a critical section guarding the set (D_s) of shared variables, then a memory system is entry consistent if it satisfies the following three conditions:

1. *An acquire access of s is not allowed to perform with respect to a processor p_i until all updates to D_s have been performed with respect to p_i.*
2. *Before an exclusive mode access to s by a processor p_i is allowed to perform with respect to p_i, no other processor may hold s in non-exclusive mode.*
3. *After an exclusive mode access to s has been performed by processor p_i, any processor's next non-exclusive mode access*

[9] As far as software is concerned, a LRC memory and a RC memory exhibit the same behavior, i.e., no modifications are needed to execute a program written for a RC memory on a LRC memory and vice versa.

to s may not be performed until it is performed with respect to p_i.

Some important differences exist between RC and EC. First, RC requires that all updates to any shared data must be performed before a release is performed. This includes data not guarded by the just released synchronization variable. On the contrary EC only requires that accesses guarded by the synchronization variable being released be performed remotely and only when the variable is next acquired by a remote processor. A release is therefore used to indicate that a synchronization variable is free and can be granted to another processor.

Second, RC requires that all previous accesses by a processor for synchronization variables be performed with respect to all other processors before that processor's shared access is observed by any other processor. In contrast EC only requires that the acquire access for the synchronization variable which guards the data being accessed be performed.

Chapter 2

Process Mobility and Memory Consistency

Because MOS provides at the messenger level, a view of memory consisting of data items without explicit address, it is natural to provide at this level a data-based DSM system. This is a variation of non-page-based DSM systems that adapts its unit of sharing to correspond to data structures provided by the programming language. The main advantage is reduced false sharing; however integration with virtual memory becomes difficult to achieve.

Among the various memory consistency models, entry consistency (EC) seems to be the best memory model candidate for DSM at the messenger level because (a) its unit of sharing is based on data structures provided by the programming language and therefore matches well the messenger view of memory and (b) it guarantees that the memory system exhibits a sequentially consistent behavior for properly labeled or data-race free programs.

Unfortunately, EC and all other weakly ordered systems (RC, RLC and WC) seem not to work properly in environments where processes can move among different nodes (processors) as is the case for messengers. As with these systems, memory is made consistent only at synchronization points, a process acquiring a lock or any other memory synchronization variable is restricted to execute its critical section entirely on the node where it acquired the lock or the variable.

To illustrate the behavior of the different weakly ordered memory systems—the state of the art memory systems—let us consider the following scenario. We have two nodes N_1 and N_2 and one process p. Each node contains in its local memory a copy of all shared variables. Process p executes a sequence of instructions that access two locations x and y in shared memory (see program 2.1). The *moveto* instruction has the effect of moving the process to the specified node, i.e., the instructions following the *moveto* are executed on the new node. We assume that s is a synchronization variable that protects the variables x and y. We assume also that memory is consistent when process p starts and that memory locations x and y contain the value 0.

```
1 acquire(s)
2    w(x)5;              store 5 in x
3    moveto(N2);
4    a := r(x);          store the value of x in register a
5    w(y)(a+10);         store (a+10) in y
6 release(s)
```

Program 2.1: Instructions executed by process p.

As far as process p is concerned, access to shared memory occurs properly inside a critical section. As a consequence, p should expect a sequentially consistent behavior of the memory system inside the critical section. When p leaves the critical section, it expects to have written the value 5 in memory location x and the value 15 in memory location y. That would be the case if the sequence of instructions did not contain the *moveto* instruction. Here after, we show that various weakly ordered DSM systems fail to achieve sequential consistency in the critical section for process p. Not surprisingly, these memory consistency models have been designed for implementation in hardware in multiprocessor systems. Multiprocessor systems are primarily used to boost the performance of parallel applications. An application is decomposed in as many threads as the number of available processors. Each thread executes on a dedicated processor, and the different processors use shared memory for data exchange between the different threads. The principal aim of relaxed memory consistency models in this arena is to allow optimizations that violate sequentially consistency in order to hide memory access latency. Commonly used optimizations relax program order to allow execution sequences that are not SC. The situation here is quite different. The system is composed with a number of general purpose workstations linked through an unreliable network. Each node (workstation) can execute concurrently a number of processes. And processes can move from one node to another. The aim of consistency models here is to offer a shared memory abstraction in software in an efficient way. Following are the behaviors of various weakly ordered memory systems with respect to the above scenario.

Firstly for EC, the memory system ensures that process p gets updated values of x and y when it acquires the lock s. Memory location x is modified on node N_1 and afterwards p moves to node N_2. The next read from location x will read the value on node N_2 (0) which has not been yet updated because node N_2 has not yet acquired the lock s. For an EC memory system, no synchronization occurs at this moment at this stage. Thus, the next write will store the value $(0 + 10)$ in location y. Next the process releases the lock s. At this stage, the two nodes contain different values in locations x and y. Next if another process p_1 executing, let us say, on node N_3, acquires the lock s just after it has been released by p, the memory system will update the memory locations x and y of N_3 from the node which last acquired the lock s,[1] i.e., node N_1. The values of

[1] One can also assume that memory is updated from the node which last released the lock.

x and y are not updated on other nodes. Hence process p_1 will see completely different values from what p expected; namely 5 for x and 0 for y (see figure 2.1).

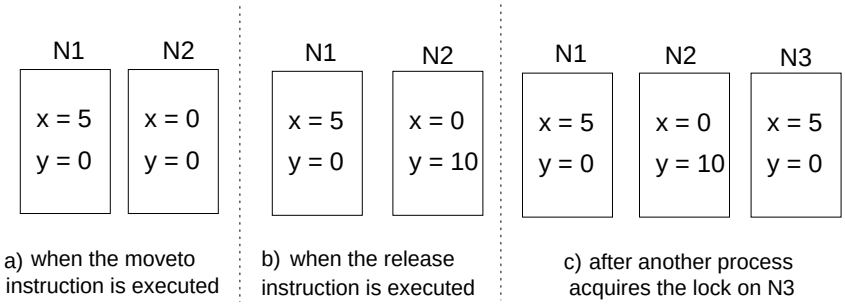

Figure 2.1: State of an EC shared memory when p is executed.

Secondly, for an RC memory system, shared memory is updated at release time. As the release is executed on node N_2, we assume that shared memory is updated from node N_2.[2] Under RC, all the copies of memory locations x and y are updated on all the hosts. The state of the memory system for this case is given in figure 2.2.

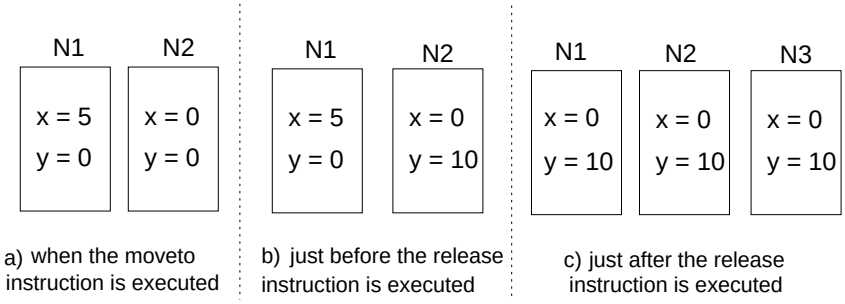

Figure 2.2: State of an RC shared memory when p is executed

The same scenario shows that the other weakly ordered memory systems—the state of the art memory systems—also fail to handle properly process mobility. In other words, a properly labeled program will see a sequentially consistent behavior from a weakly ordered memory system, only if the program satisfies a further constraint; namely *a process cannot move from one node to another when it is inside a critical section.* We present in the following subsection a new memory consistency model that we call *access consistency* and that is well suited for a messenger environment.

In this case, memory locations x and y would be updated from node N_2 and would contain respectively 0 and 10, what still is different from what expected.

[2]One can also assume that shared memory is updated from the host that last acquired the lock. In this case, x and y would be updated from node N_1 and would contain respectively the values 5 and 0 on all the nodes.

Chapter 3

Access Consistency

The main idea behind access consistency (AcC) is to update a shared data item only when a process tries to access it. Only the data item being accessed by the process is updated. One can observe the similarity with the behavior of some OSes in regard to memory management. A page belonging to a process is brought in main memory only when it is requested by the process.

Because an AcC memory system guarantees to update a data item whenever it is requested by a process, the process does not need to acquire a lock before accessing the data item in order to ensure that it gets an up-to-date value. As a consequence, a process will always get the "right" value even if it has moved to another node (processor).

For any running process, an AcC memory achieves an atomic view of any individual access to shared memory. However, a process that needs an atomic view for a sequence of accesses to shared memory must protect that sequence in a critical section by a means completely decoupled from the memory system, for example by acquiring the lock protecting the data items that are accessed in the sequence. This conforms to the memory behavior in a system without shared memory. In fact, in weakly ordered memory systems, a lock protecting a critical section is used also to protect the memory locations accessed inside the critical section. As a consequence, a memory item cannot be updated from another node inside a critical section, and if the process executing the critical section moves to another node, the memory fails to achieve sequential consistency inside the critical section.

More precisely, AcC assumes the following conditions are met:

- *accesses to distributed shared memory can be distinguished from other accesses to memory;*
- *a read access to shared data can be distinguished from a write access to shared data.*

Then, a memory system is AcC consistent if it satisfies the following conditions:

Condition 3.1 For each memory location m, all processors agree on the order of all operations on m although any two processors can see writes to different memory locations in a different order.

Condition 3.2 A read operation r from memory location m is not allowed to perform with respect to processor p_i until the most recent write access to m relative to r has been performed with respect to p_i.

Condition 3.3 A write access w to memory location m is not allowed to perform with respect to processor p_i until all most recent accesses to m relative to w have been performed with respect to p_i.

Condition 3.1 defines a total order on all operations on a given memory location (more details on this order are provided below). We will use $<$ to note that order. Thus we will write $o_1^m < o_2^m$ if operation o_1^m on memory location m occurs before operation o_2^m effected on the same memory location. Then we have the following definitions:

Definition 3.1 *Two operations o_1^m and o_2^m on the same memory location m are concurrent operations if and only if (a) o_1^m and o_2^m are read operations from different processors and (b) there does not exit a write operation w^m such that ($o_1^m < w^m < o_2^m$ or $o_2^m < w^m < o_1^m$).*

Definition 3.2 *A write operation w^m to memory location m is the most recent write access to m relative to an operation o^m (read or write) if $\forall$ operation o_i^m such that $w^m < o_i^m < o^m$ then o_i^m is a read operation.*

A read access r^m to memory location m is a most recent read access to m relative to a write operation w^m if (a) $\forall$ operation o^m such that $r^m < o^m < w^m$ then o^m is a read operation and (b) r^m and o^m have not been issued by the same processor.

A most recent access to memory location m relative to a write operation w^m is either the most recent write to m relative to w^m or a most recent read to m relative to w^m.

It appears from the preceding definitions that there exists only one most recent write operation relative to any operation. However, there may exist more than one most recent read operations relative to a write operation. In fact, if r^m is a most recent read operation relative to w^m, then any concurrent read operation to r^m is also a most recent operation relative to w^m.

Condition 3.1 states that all accesses to a given location are totally ordered. The order required for AcC by condition 3.1 is more "weaker" than that specified by SC. The order required by AcC—we will call it AcC order—must preserve program order; furthermore, the different processors must agree on the order of memory accesses on each memory location. However AcC order is somehow

"weak" in the sense that two execution sequences[1] are considered identical even if they do not preserve the order of concurrent reads. This means that if r_1^m and r_2^m are two concurrent reads, a sequence where r_1^m appears before r_2^m is considered to be identical to a sequence where r_2^m appears before r_1^m provided that the order of all other non concurrent operations is preserved. In other words, if r_1^m and r_2^m are concurrent we can assume indifferently that $r_1^m < r_2^m$ or $r_2^m < r_1^m$.

Actually, AcC does not require each processor to see all the operations on a memory location m issued by other processors. It is sufficient that each processor sees from other processors, only operations on memory location m that are most recent relative to its own operations on m. Therefore updates to shared data items can be effected on demand from each processor.

Condition 3.2 ensures that each read operation returns the value written by the most recent write operation at the same location. It also states that concurrent reads can be performed on a memory location. Finally, condition 3.3 specifies that write operations on a memory location are exclusive, i.e., they cannot be performed concurrently with any other operation. However, writes on different memory locations can occur concurrently.

After having defined precisely what an AcC memory system is, let us see how it behaves when we consider the same scenario described above (see program 2.1). AcC does not relate locks to memory operations, therefore, when process p executes the `acquire(s)` instruction, nothing special occurs (in terms of memory operations to ensure consistency). Here, as far as the process p is concerned, the bracket consisting of the `acquire(s)/release(s)` pair of instructions ensures that execution of the bracketed instructions appears atomic with respect to accesses to memory locations x and y. As for the weakly ordered memory systems, memory location x is updated with value 5 on node N_1 and then process p moves to node N_2. Next, when process p reads the value of x on node N_2, x is updated with the value 5 that p stored in that location on node N_1 (because no other process has modified[2] x and AcC ensures that a data item is updated whenever it is accessed by a process). As a result, p will write the value $(5 + 10)$ in memory location y before releasing the lock s. However, the value of y is updated only on node N_2. It will be updated on any other node whenever a process executing there will try to access y. This results in the SC behavior expected by process p; the next process to access x will see the value 5; and the next process to access y will get the value 15 (see figure 3.1) because these are the values written in x and y respectively by the most recent write operations on those memory locations.

From the above conditions for AcC consistency, it trivially follows that SC is strictly stronger than AcC, i.e, any SC execution is also AcC, however, an AcC execution may not be SC. For example, by considering only condition 3.1, one

[1] An execution sequence is a sequence of accesses to memory as perceived by an individual processor. The sequence of accesses perceived by any processor P_i may be different from that perceived by another processor P_j.

[2] The `acquire(s)` ensures that x and y are accessed in a critical section. We also assume that any access to these variables requires the acquisition of s.

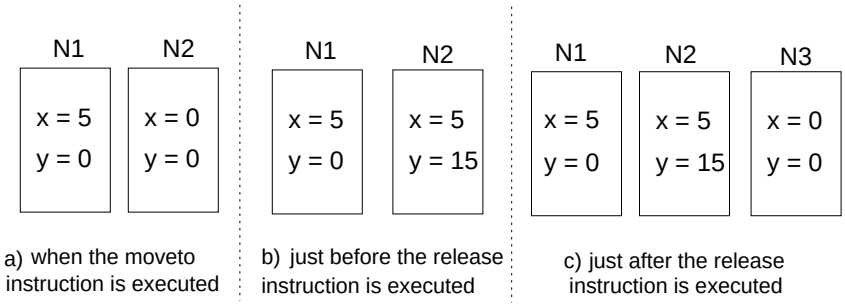

Figure 3.1: State of an AcC shared memory when p is executed.

sees that AcC allows execution sequences that are not allowed by SC. Indeed, AcC allows operations on different memory locations to occur concurrently. Moreover condition 3.2 makes it possible to execute concurrently read operations on the same location. This relaxes further the constraints of SC on a memory system. As a consequence, we can expect AcC to have better performance than SC.

Both EC and AcC do not rely on a pre-established (fixed) unit of sharing, on the contrary, they adapt their sharing unit to match the data structures being accessed. However, some fundamental differences exist between the two memory models. Firstly, EC distinguishes synchronization variables from other shared data. Accesses to synchronization variables must be distinguished from accesses to other variables; programs running on an EC memory system use the `acquire` and `release` operations to access synchronization variables and normal read and write operations to access other shared data. Moreover, EC guarantees that only shared data accessed inside critical sections contain coherent information. On the contrary, AcC does not provide the notion of synchronization variable. But AcC requires that (a) accesses to shared data be distinguished from accesses to other data and (b) among accesses to shared data, read operations be distinguished from write operations. And AcC ensures that a shared data item is updated whenever a process accesses it, therefore, a process always gets an up-to-date value for a shared data item.

Secondly, in an EC memory system, a synchronization variable is used to protect a set of shared data items. And the association between the protected data and the synchronization variable is allowed to change dynamically under the control of the software executing on the memory system. While this flexibility seems a priori appealing, it can however be a source of bugs. AcC ensures only that each individual access to a shared data item is "atomic" for the process accessing it. It does not provide any means to protect a set of shared data items in order to get an atomic behavior for a sequence of accesses to the protected data. Software must use mechanisms provided by programming languages, the run time environment or the operating system in order to achieve such atomic behavior.

And finally, we showed through a simple scenario that EC, and the other weakly ordered memory systems fail to handle process mobility. AcC has been defined especially to handle process mobility in a messenger environment.[3]

3.1 Implementing AcC

From the definition of AcC consistency, it follows that both read and write operations are blocking operations (conditions 3.2 and 3.3). A read operation cannot return until the most recent write operation has performed, and a write operation cannot return until all most recent operations are performed. Consequently, a straight forward implementation of AcC can associate a lock with each shared data item. Each operation on the data item first acquires the associated lock, performs the intended operation and next releases the lock.

More precisely, each shared data item (data structure at the language level) is associated the following information:

- Owner: the owner of a shared data item is the processor that performed the last write operation on the data item or the processor that created the data if a write operation has not yet been performed on the data.
- Lock and processor list: a lock on a data item can be held by a processor either in read-only mode in which case the lock can be shared with other processes, or in read-write mode in which case the lock is exclusive. Therefore only the data owner can hold an exclusive lock on the data item. Furthermore, the data owner maintains a list of processors holding a read-only lock on the data.
- Request list: the processor will maintain a FIFO list of requests from other processes to access the data item.

Each processor[4] services the requests on data items on a first in first out basis. When a process performs a request, it is blocked (the request is put on the data FIFO list) until the processor terminates serving the request (the request is removed from the data FIFO list), upon which the process is unblocked and can issue another memory access request.

Then when a processor finds a read request in the FIFO associated to a shared data item, it checks if it holds a lock on the data item. If so, the processor returns the local value of the data to the process. On the contrary, the processor cannot be the data owner (a data owner has always either a read-only or a read-write lock on the data); it performs the following sequence of operations:

[3] AcC is well suited for any environment sharing the properties of a messenger environment, namely: (a) support for process mobility, (b) possibility to distinguish efficiently shared data accesses from other accesses (this can be done easily in interpreted environments) and (c) possibility to use efficiently data structures provided by the programming language as the unit of sharing.

[4] Here we use the terms node and processor to refer to the shared memory software on the node.

1. the processor locates the data owner;
2. the processor requests a read-only lock on the data to the data owner and waits for the response. The response will contain also the up-to-date value of the data item;
3. the processor stores the received value in the local copy of the data item;
4. the processor returns the received value to the process and the process resumes execution.

When a processor finds a write operation on a shared data item, it checks if it holds an exclusive lock on the data (in this case, the processor is the data owner). If this is the case, the processor updates the local copy of the data and the process resumes execution. On the contrary:

1. if the processor is not the data owner, it locates the data owner;
2. if the processor is not the data owner, it requests data ownership from the current data owner and waits from the response. The response will contain also the list of processors holding a read-only lock on the data;
3. the processor requests each host holding a read-only lock on the data to release the lock and waits for the response;
4. the processor acquires an exclusive lock on the data;
5. the processor updates the local copy of the data with the value provided by the process and the process resumes execution.

Each request from one processor to another to acquire a lock on a data item or to acquire data ownership contains a timestamp (such requests are addressed only to a data item owner), for example one generated using Lamport logical clocks []. The data owner maintains a list of those requests and uses the timestamps to handle the requests by processing first the older requests. And when ownership is transferred to another processor, the list of remaining requests is also transferred to the new data owner. Because for each data item, there is only one data owner, there will be only one list of requests for the data item. The data owner examines the list of requests after execution of each memory operation, and executes the oldest request found (provided the list is not empty). Then the data owner switches back to the execution of the next memory operation requested by a running process. In this way, all requests on a given data item are serviced fairly.

On a request from another processor to acquire data ownership for a data item, the data owner releases the lock it holds on the data and transfers to the requester (a) the list of processors holding a read-only lock on the data and (b) the list of requests from other processors remaining to be handled. On a request from another processor to acquire a read-only lock on a data item, the data owner adds the requesting processor on the list of hosts holding a read-only

lock on the data and sends to the requester a copy of the local data value. Then the data owner degrades the lock it holds on the data from read-write to read-only. And finally, if a processor holding a read-only lock on a data item receives from another processor a request to release the lock, the processor releases the lock and transfers it (sends an acknowledgement) to the requester.

3.2 The given implementation achieves AcC

In this subsection, we give an informal proof of the fact that the implementation proposed above actually achieves AcC. It is sufficient to consider two cases: (a) all the processes execute on the same processor and (b) the processes execute on different processors. For each case we show that the implementation satisfies the three conditions for AcC (Condition 3.1 through Condition 3.3).

Firstly, for the case where all the processes execute on the same processor, the FIFO associated to a data item serializes the accesses to the data item and therefore insures that both processes see all the operations on the data in the same order. Furthermore, the fact each process is blocked until its requested memory operation is performed by the processor ensures that the memory system preserves program order. As there is only one processor, it will be the owner of all shared data items. Then conditions 3.2 and 3.3 trivially follow from the fact that all accesses to a shared data item are serialized by its associated FIFO. Indeed, no operation is allowed to be performed on the data item until the preceding operation (in the FIFO) has been performed.[5]

Next, for the case where the processes execute on different processors, a given data item is owned by only one host. The owner is allowed to perform read and write operations on the data item. To show that condition 3.1 holds, we construct a global sequence of operations performed on a data item as follows:

- The owner of a data item adds each read operation performed on the data item on the global sequence of operations: whenever the execution of a read operation is finished, the read is added on the global sequence of operations;
- Each host holding a read-only lock on the data item maintains a local sequence of read operations performed on the data item; the reads are added in the local sequence of reads in the order they are executed by the host;
- Before adding a write operation on the global sequence of operations, the owner data owner collects the partial (local) sequences of reads requests effected by other processors as follows. The implementation requests that a processor acquires an exclusive lock on a data item before performing a write operation on the data. Two cases can arise:

[5] The implementation is even stricter than AcC which allows concurrent reads to be performed concurrently.

1. The processor performing the write operation has already an exclusive lock on the data (and is therefore the data owner): the processor adds the write operation on the sequence of operations. No other operation has been performed on the data by another processor.
2. The processor performing the write operation has not yet an exclusive lock on the data. In this case, the implementation requests that the processor acquire data ownership before locking the data. At acquire time, the processor receives from the data owner (if it is not the data owner), the list of hosts holding a read-only lock on the data item. Then the processor requests that each processor holding a read-only lock on the data relinquish it. At this stage, any process holding such a lock sends an acknowledgement together with its local sequence of read operations performed on the data item and the processor reinitializes the local sequence of read operations to an empty sequence. The processor performing the write operation adds the received local read sequences on the global sequence of operations performed on the data item.

   In this way, all read operations performed by different processors between any two consecutive write operations are added on the global sequence of operations performed on the data. As the reads from each processor maintain program order, they will maintain program order in the global sequence of operations performed on the data item. The processor performing the write operation does not need to know in which order to add the different local read sequences on the global sequence of operations because all such reads are concurrent and are therefore considered to be performed in any order. After having added all the local read sequences on the global sequence of operations, the processor proceeds performing the write operation and adds the write on the global sequence of operations on the data item.

- Whenever data ownership is transferred to another host, the host transferring the data ownership transfers also to the new data owner, the global sequence of operations performed so far on the data item.

The global sequence of operations on a data item built with the above algorithm preserves program order and is seen by all the processors.

Conditions 3.2 and 3.3 follow from the fact that a node performing a read operation on a data item must first acquire a read-only lock on the data and a node performing a write operation must first acquire an exclusive lock on the data. Indeed, lock acquisitions are serialized by the data owner which handles the requests one at time. And acquiring an exclusive lock on a data item ensures that all the previous operations on the data have been performed (because all read-only locks and the previous read-write lock on the data are discarded and no operation can be performed if a lock is not held). Similarly, acquiring a read-only lock on a data item discards the write-only lock held on the data and

therefore ensures that the previous write on the data has completed.

3.3 Data Item Ownership

The proposed implementation does not specify how the owner of a data item is located. Either a centralized or a distributed approach can be used for that purpose. One centralized approach would be to have one node maintain the list of data owners and play the role of a server (manager). Each request to acquire a lock will be addressed to the server that relays the request to the data owner. The data owner handles the request and addresses the response directly to the node that requested the lock. Acquisition of data ownership could be handled as follows: (a) the requester sends the request to the server and waits for the response, (b) the server forwards the request to the data owner and waits for the response, (c) the owner handles the request and sends the response to the server and (d) the server updates ownership information and forwards the response to the requester which becomes the new data owner.

The problem with the centralized solution is that the server becomes a bottleneck in the system. Furthermore, failure of the server will certainly be costly to the system in the sense that either all the system will fail (if it is not fault tolerant) or the recovery process will be very long. A classical approach to relax the burden imposed on the "one server" is to partition the data and distribute it among the different nodes, i.e, to have one node be the server for a set of data items only. In this way, we expect that the load is distributed between the different servers. Moreover, any failure of a single system should not affect the whole system and, in a fault tolerant system, the recovery process is expected to be faster because there is less data to process for any single node.

Another alternative is to modify a little the proposed implementation. Instead of having data ownership changing from one node to another, the owner of a data item would be fixed, for example it would be the node that created the data item. Now, to perform a write operation on a data item, a processor does not acquire data ownership on the item, instead it requests only from the data owner an exclusive read-write lock on the data. Update to the data is performed only after the lock has been received. And to perform a read operation, a node requests, as in the proposed implementation, a read-only lock on the data to the data owner. This approach seems to solve the bottleneck problem present in the "on server" approach provided that data items are well distributed on different nodes (if all data items are created by the same node, we have a degenerated case similar to the one server approach). However, this approach is not well suited for an up/down implementation in a dynamic environment. Indeed ...

A decentralized approach in locating a data item owner would be to have each node maintain a record of the data probable owner which is the best guess for the owner of the data. The probable owner is initialized with the node that created the data item. Each node addresses each request (lock and ownership acquisition) for a data item to the data probable owner. And any node that receives a request for a data item for which it is not the owner, forwards the

request to its own probable data owner. Whenever a node transfers ownership to another node, the former owner updates its probable owner field with the new data owner. This ensures that a request on a data item follows a chain of probable owners until it reaches the actual owner of the data. However, the chain of probable owners may grow with the requests to acquire data ownership.[6] In this case, a request has to be forwarded many times to reach the actual data owner. Given the fact that sending a message in a network of workstations has a high cost, this scheme may become very costly in large networks.

A simple variation of the probable owner approach is to have the true owner of a data item send periodically information to other nodes to allow them update their best guess of the data owner.[7] This has the benefit of preventing a sequence of probable owners to grow very long in a large environment. Its drawback is that the information is sent to a large number of nodes including the hosts that have the correct guess for the data owner, and therefore requires a big number of messages to be sent (we assume that the network does not support broadcasting and multicasting). Moreover, it is not easy to determine the most appropriate time to send update information to other hosts. Indeed, on one side, waiting longer than necessary before sending such information may not be effective because the number of requests that follow a long chain will potentially be increased. On the other hand, sending the update information earlier than necessary can have little impact if most nodes have the best guess for the data

[6] In a network of N nodes, a request has to be forwarded at most $N-2$ times. We do not consider the first message sent by the requester as a forward message. Therefore, in the worst case, $N-1$ messages are sent to reach the data owner. We can consider $N/2$ to be the average number of messages sent to reach the data owner.

We have assumed in the above approximations that only one request to acquire data ownership for a given data item circulates in the network. When multiple requests to acquire ownership can be pending, the number of messages needed to reach the data owner in the worst case can be as much as $(N-1)+(N-2) = 2N-3$. This is the case when all the nodes on the chain followed by the request have also requested data ownership. It is then possible that all the requests reach the data owner and are all queued before the first request is serviced. If the requests are queued in the reverse order of the nodes forming the chain, ownership will also be transferred in the reverse order. The request originated from the outer host (the first host in the chain) will be forwarded until it reaches the host at the end of the chain. This will necessitate $N-1$ messages. The first request is then serviced and ownership is transferred to the preceding node in the chain, together with the queue of pending requests. The request originated from the first node in the chain will be forwarded $N-2$ times in the reverse order. Therefore a total of $2N-3$ messages is needed to reach the node that will handle the request.

It is therefore possible to use the following optimization: a node that receives a request to transfer data ownership forwards the request to its best guess only if the node itself has not yet issued a request to acquire data ownership. On the contrary if the node has already issued a request to acquire data ownership, it queues the received request. This request will be handled after the node receives data ownership. At that time, either the request is put on the queue of pending requests if there are pending requests, or it is handled by the new data owner. To maintain fairness, the timestamps are used to handle first the oldest requests.

[7] This variation uses the following simple heuristics: *the cost of reaching the owner of a data item increases with time.* As time passes, the probability that the owner of a data item changes, increases and therefore, the chain to reach the data owner becomes longer. Note that the optimization that consists of not allowing a host to forward a request for a data item for which it has itself already requested ownership can be applied here.

owner. The cost of sending update messages is not amortized.

Another variation of the probable owner approach allows a selective update of the best guess according to the requests to acquire data ownership. Whenever a request to acquire data ownership, originated from one node, reaches the data owner, information to update the best guess for data ownership is sent back on the reserve path followed by the request.[8] In this way, all the nodes on the chain followed by the request will update their best guess for the data owner in the reverse order in which the request reached them.

3.4 Discussion

From the above implementation, it follows that AcC can be seen as a degenerated case of EC where each shared data item is protected by its own synchronization variable and where a synchronization variable cannot protect more that one shared data item. When a processor accesses a data item, it first acquires the unique lock associated to the data, then it updates the data as required and after that, it releases the lock. Acquiring or releasing the data lock is not requested explicitly by a process, instead, it is performed automatically by the memory system on each access, in a way completely transparent to the process.

We assumed in the implementation that each shared data item is completely replicated on all the nodes. This allows read operations to be performed concurrently and at the same time reduces the cost of read operations while making write operations more costy. Indeed, the local copy of a data item is usually used in read operation (when the node performing the read holds already a lock on the data); for a write operation all read-only locks held on the data by other nodes have to be discarded. Furthermore, the amount of memory consumed to represent a shared data item is very high. Beside the amount of memory needed to represent the data value itself, additional memory is required to represent the information needed by the memory system to handle operations on the data item (the data owner, the lock, the list of hosts having read-only locks on the data, the queue of requests pending on the data, etc). The amount of memory needed to represent this additional information does not depend on the size of the shared data item. As a consequence, big shared data items will have less memory overhead[9] than smaller shared data items. In order to reduce the memory overhead, one can group an number of shared items to create a bigger data structure. The so created data structure becomes the unit of sharing for the memory system and not the individual items forming the data structure.[10] However doing so increases the potential of false sharing, especially

[8] The optimization consisting of not forwarding a request if the node has itself requested data ownership can be applied in this case also.

[9] Here memory overhead represents the ratio between the additional amount of memory needed to represent a data item and the actual amount of data needed to represent the value of the shared data item. Replicating a shared data item on N nodes multiplies the data overhead by N. However replicating is an admitted solution when one compares the cost of memory to the benefit of speeding up shared memory operations.

[10] In MØ, the programmer can group a number of unrelated data items in a dictionary and

when unrelated items are put together. One looses the principal benefit of a data-based-shared memory system when the size of shared objects increases, especially when unrelated data items are put together to form a bigger data structure.

When a number of processes running on different processors are heavily sharing a data item, and if the proportion of write operations is very high compared to that of read operations, the data ownership will frequently change from one processor to another. It is even possible to reach the degraded situation where each node sits in a loop where it acquires data ownership, performs only one memory operation on the data item namely an update operation, relinquishes data ownership and then requests again data ownership to perform the following memory operation on the data. This situation is similar to the situation which arises in a weakly ordered memory system when many processes are heavily contending for acquiring a lock that protects shared data. In order to enhance the performance of an AcC memory system in this situation of high contention, it is necessary to group in critical sections as much number of accesses to the shared data item as possible. This will allow a number of accesses to be performed on the shared data by a node before relinquishing data ownership for the benefit of another node. However such grouping of memory operations is not always easy to achieve, it may even be impossible to do.

make the dictionary the shared object.

Bibliography

[ABJ+93] Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. Technical Report GIT-CC-92/34, College of Computing, Georgia Institute of Technology, March 1993.

[BZ91] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University (CMU), September 1991.

[BZS93] Brian N. Bershad, Matthew J. Zekauskas, and W. A. Sawdon. The midway distributed shared memory system. In *Proceedings of the IEEE COMPCOM Conference*, pages 528–537. IEEE, 1993.

[DMMTH95] Giovanna Di Marzo, Murhimanya Muhugusa, Christian F. Tschudin, and Jürgen Harms. The Messenger Paradigm and its Impact on Distributed Systems. In *ICC'95 workshop on Intelligent Computer Communication*, 1995.

[DSB86] Michel Dubois, Christoph Scheurich, and Fayé A. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442. ACM, 1986. Weak Consistency.

[DSB88] Michel Dubois, Christoph Scheurich, and Fayé A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. In *IEEE Computer*, volume 21, pages 9–21, February 1988.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26. ACM, 1990.

[Goo89] J. R. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, 1989.

[IDFL96] Liviu Iftode, Cezary Dubnicki, Edward W. Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, February 1996.

[KCZ92] Peter Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21. ACM, 1992.

[Kel95] Peter Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Rice University, January 1995.

[KNA93] Prince Kohli, Gil Neiger, and Mustaque Ahamad. A characterization of scalable shared memories. Technical Report GIT-CC-93/04, College of Computing, Georgia Institute of Technology, Atlanta, January 1993.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979. Sequential consistency.

[LS88] Richard J. Lipton and Jonathan S. Sandberg. Pram: A scalable shared memory. Technical Report 180-88, Departement of Computer Science, Princeton University, 1988.

[Mis86] Jayadev Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transaction on Programming Languages and Systems*, 8(1):142–153, 1986. Memory Model.

[Mos93] David Mosberger. Memory consistency models. Technical Report TR 93/11, Department of Computer Science, University of Arizona, 1993.

[MRZ95] M. Mizuno, M. Raynal, and J. Z. Zhou. Sequential Consistency in Distributed Systems: Theory and Implementatio. Technical Report 2437, INRIA, INRIA Rennes, March 1995.

[Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

[TDMMH94] Christian F. Tschudin, Giovanna Di Marzo, Murhimanya Muhugusa, and Jürgen Harms. Messenger-based Operating Systems. Technical Report No 90 (Cahier du CUI), University of Geneva, 1994.

[Tsc93] Christian F. Tschudin. *On the Structuring of Computer Communications*. PhD thesis, Université de Genève, 1993. Thèse No 2632.

[Tsc94] Christian F. Tschudin. An Introduction to the M0 Messenger Language. Technical Report No 86 (Cahier du CUI), University of Geneva, 1994.