CHAPTER 5

Distributed Shared Memory

5.1 INTRODUCTION

In Chapter 3, it was mentioned that the two basic paradigms for interprocess communication are as follows:

- Shared-memory paradigm
- Message-passing paradigm

Message-passing systems (described in Chapter 3), or systems supporting Remote Procedure Calls (RPCs) (described in Chapter 4), adhere to the message-passing paradigm. This paradigm consists of two basic primitives for interprocess communication:

Send (*recipient, data*)
Receive (*data*)

The sending process generates the *data* to be shared and sends it to the recipient(s) with which it wants to communicate. The recipient(s) receive the *data*.

This functionality is sometimes hidden in language-level constructs. For example, RPC provides automatic message generation and reception according to a procedural specification. However, the basic communication paradigm remains the same because the communicating processes directly interact with each other for exchanging the shared data.

In contrast to the message-passing paradigm, the shared-memory paradigm provides to processes in a system with a shared address space. Processes use this address space in the same way they use normal local memory. That is, processes access data in the shared address space through the following two basic primitives, of course, with some variations in the syntax and semantics in different implementations:

data = **Read** (*address*)
Write (*address*, *data*)

Read returns the *data* item referenced by *address*, and *write* sets the contents referenced by *address* to the value of *data*.

We saw in Chapter 1 that the two major kinds of multiple-instruction, multiple-data-stream (MIMD) multiprocessors that have become popular and gained commercial acceptance are tightly coupled shared-memory multiprocessors and loosely coupled distributed-memory multiprocessors. The use of a shared-memory paradigm for interprocess communication is natural for distributed processes running on tightly coupled shared-memory multiprocessors. However, for loosely coupled distributed-memory systems, no physically shared memory is available to support the shared-memory paradigm for interprocess communication. Therefore, until recently, the interprocess communication mechanism in loosely coupled distributed-memory multiprocessors was limited only to the message-passing paradigm. But some recent loosely coupled distributed-memory systems have implemented a software layer on top of the message-passing communication system to provide a shared-memory abstraction to the programmers. The shared-memory abstraction gives these systems the illusion of physically shared memory and allows programmers to use the shared-memory paradigm. The software layer, which is used for providing the shared-memory abstraction, can be implemented either in an operating system kernel or in runtime library routines with proper system kernel support. The term Distributed Shared Memory (DSM) refers to the shared-memory paradigm applied to loosely coupled distributed-memory systems [Stumm and Zhou 1990].

As shown in Figure 5.1, DSM provides a virtual address space shared among processes on loosely coupled processors. That is, DSM is basically an abstraction that integrates the local memory of different machines in a network environment into a single logical entity shared by cooperating processes executing on multiple sites. The shared memory itself exists only virtually. Application programs can use it in the same way as a traditional virtual memory, except, of course, that processes using it can run on different machines in parallel. Due to the virtual existence of the shared memory, DSM is sometimes also referred to as *Distributed Shared Virtual Memory* (*DSVM*).

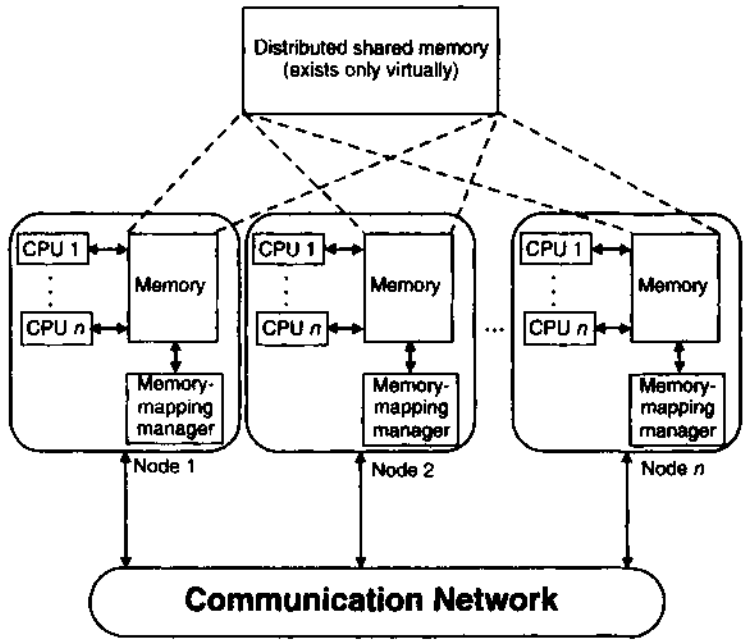

Fig. 5.1 Distributed shared memory (DSM).

5.2 GENERAL ARCHITECTURE OF DSM SYSTEMS

The DSM systems normally have an architecture of the form shown in Figure 5.1. Each node of the system consists of one or more CPUs and a memory unit. The nodes are connected by a high-speed communication network. A simple message-passing system allows processes on different nodes to exchange messages with each other.

The DSM abstraction presents a large shared-memory space to the processors of all nodes. In contrast to the shared physical memory in tightly coupled parallel architectures, the shared memory of DSM exists only virtually. A software memory-mapping manager routine in each node maps the local memory onto the shared virtual memory. To facilitate the mapping operation, the shared-memory space is partitioned into *blocks*.

Data caching is a well-known solution to address memory access latency. The idea of data caching is used in DSM systems to reduce network latency. That is, the main memory of individual nodes is used to cache pieces of the shared-memory space. The memory-mapping manager of each node views its local memory as a big cache of the shared-memory space for its associated processors. The basic unit of caching is a memory block.

When a process on a node accesses some data from a memory block of the shared-memory space, the local memory-mapping manager takes charge of its request. If the memory block containing the accessed data is resident in the local memory, the request is satisfied by supplying the accessed data from the local memory. Otherwise, a network block fault is generated and the control is passed to the operating system. The operating

system then sends a message to the node on which the desired memory block is located to get the block. The missing block is migrated from the remote node to the client process's node and the operating system maps it into the application's address space. The faulting instruction is then restarted and can now complete. Therefore, the scenario is that data blocks keep migrating from one node to another on demand but no communication is visible to the user processes. That is, to the user processes, the system looks like a tightly coupled shared-memory multiprocessors system in which multiple processes freely read and write the shared-memory at will. Copies of data cached in local memory eliminate network traffic for a memory access on cache hit, that is, access to an address whose data is stored in the cache. Therefore, network traffic is significantly reduced if applications show a high degree of locality of data accesses.

Variations of this general approach are used in different implementations depending on whether the DSM system allows replication and/or migration of shared-memory data blocks. These variations are described in Section 5.6.1.

5.3 DESIGN AND IMPLEMENTATION ISSUES OF DSM

Important issues involved in the design and implementation of DSM systems are as follows:

1. *Granularity.* Granularity refers to the block size of a DSM system, that is, to the unit of sharing and the unit of data transfer across the network when a network block fault occurs. Possible units are a few words, a page, or a few pages. Selecting proper block size is an important part of the design of a DSM system because block size is usually a measure of the granularity of parallelism explored and the amount of network traffic generated by network block faults.

2. *Structure of shared-memory space.* Structure refers to the layout of the shared data in memory. The structure of the shared-memory space of a DSM system is normally dependent on the type of applications that the DSM system is intended to support.

3. *Memory coherence and access synchronization.* In a DSM system that allows replication of shared data items, copies of shared data items may simultaneously be available in the main memories of a number of nodes. In this case, the main problem is to solve the memory coherence problem that deals with the consistency of a piece of shared data lying in the main memories of two or more nodes. This problem is similar to that which arises with conventional caches [Smith 1982], in particular with multicache schemes for shared-memory multiprocessors [Frank 1984, Goodman 1983, Katz et al. 1985, Yen et al. 1985]. Since different memory coherence protocols make different assumptions and trade-offs, the choice is usually dependent on the pattern of memory access. The terms *coherence* and *consistency* are used interchangeably in the literature.

In a DSM system, concurrent accesses to shared data may be generated. Therefore, a memory coherence protocol alone is not sufficient to maintain the consistency of shared data. In addition, synchronization primitives, such as semaphores, event count, and lock, are needed to synchronize concurrent accesses to shared data.

4. *Data location and access.* To share data in a DSM system, it should be possible to locate and retrieve the data accessed by a user process. Therefore, a DSM system must implement some form of data block locating mechanism in order to service network data block faults to meet the requirement of the memory coherence semantics being used.

5. *Replacement strategy.* If the local memory of a node is full, a cache miss at that node implies not only a fetch of the accessed data block from a remote node but also a replacement. That is, a data block of the local memory must be replaced by the new data block. Therefore, a cache replacement strategy is also necessary in the design of a DSM system.

6. *Thrashing.* In a DSM system, data blocks migrate between nodes on demand. Therefore, if two nodes compete for write access to a single data item, the corresponding data block may be transferred back and forth at such a high rate that no real work can get done. A DSM system must use a policy to avoid this situation (usually known as thrashing).

7. *Heterogeneity.* The DSM systems built for homogeneous systems need not address the heterogeneity issue. However, if the underlying system environment is heterogeneous, the DSM system must be designed to take care of heterogeneity so that it functions properly with machines having different architectures.

These design and implementation issues of DSM systems are described in subsequent sections.

5.4 GRANULARITY

One of the most visible parameters to be chosen in the design of a DSM system is the block size. Several criteria for choosing this granularity parameter are described below. Just as with paged main memory, there are a number of trade-offs and no single criterion dominates.

5.4.1 Factors Influencing Block Size Selection

In a typical loosely coupled multiprocessor system, sending large packets of data (for example, 4 kilobytes) is not much more expensive than sending small ones (for example, 256 bytes) [Li and Hudak 1989]. This is usually due to the typical software protocols and overhead of the virtual memory layer of the operating system. This fact favors relatively large block sizes. However, other factors that influence the choice of block size are described below [Nitzberg and Virginia Lo 1991].

1. *Paging overhead.* Because shared-memory programs provide locality of reference, a process is likely to access a large region of its shared address space in a small amount of time. Therefore, paging overhead is less for large block sizes as compared to the paging overhead for small block sizes.

2. *Directory size.* Another factor affecting the choice of block size is the need to keep directory information about the blocks in the system. Obviously, the larger the block size, the smaller the directory. This ultimately results in reduced directory management overhead for larger block sizes.

3. *Thrashing.* The problem of thrashing may occur when data items in the same data block are being updated by multiple nodes at the same time, causing large numbers of data block transfers among the nodes without much progress in the execution of the application. While a thrashing problem may occur with any block size, it is more likely with larger block sizes, as different regions in the same block may be updated by processes on different nodes, causing data block transfers that are not necessary with smaller block sizes.

4. *False sharing.* False sharing occurs when two different processes access two unrelated variables that reside in the same data block (Fig. 5.2). In such a situation, even though the original variables are not shared, the data block appears to be shared by the two processes. The larger is the block size, the higher is the probability of false sharing, due to the fact that the same data block may contain different data structures that are used independently. Notice that false sharing of a block may lead to a thrashing problem.

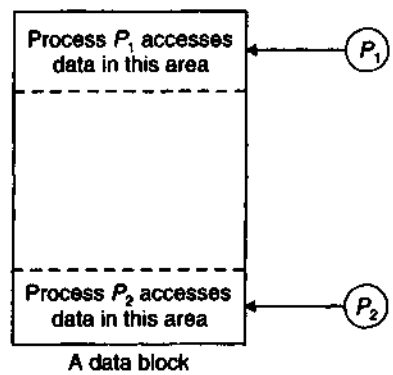

Fig. 5.2 False sharing.

5.4.2 Using Page Size as Block Size

The relative advantages and disadvantages of small and large block sizes make it difficult for a DSM designer to decide on a proper block size. Therefore, a suitable compromise in granularity, adopted by several existing DSM systems, is to use the typical page size of a conventional virtual memory implementation as the block size of a DSM system. Using page size as the block size of a DSM system has the following advantages [Li and Hudak 1989]:

1. It allows the use of existing page-fault schemes (i.e., hardware mechanisms) to trigger a DSM page fault. Thus memory coherence problems can be resolved in page-fault handlers.

2. It allows the access right control (needed for each shared entity) to be readily integrated into the functionality of the memory management unit of the system.
3. As long as a page can fit into a packet, page sizes do not impose undue communication overhead at the time of network page fault.
4. Experience has shown that a page size is a suitable data entity unit with respect to memory contention.

5.5 STRUCTURE OF SHARED-MEMORY SPACE

Structure defines the abstract view of the shared-memory space to be presented to the application programmers of a DSM system. For example, the shared-memory space of one DSM system may appear to its programmers as a storage for words, while the programmers of another DSM system may view its shared-memory space as a storage for data objects. The structure and granularity of a DSM system are closely related [Nitzberg and Virginia Lo 1991]. The three commonly used approaches for structuring the shared-memory space of a DSM system are [Nitzberg and Virginia Lo 1991] as follows:

1. *No structuring*. Most DSM systems do not structure their shared-memory space. In these systems, the shared-memory space is simply a linear array of words. An advantage of the use of unstructured shared-memory space is that it is convenient to choose any suitable page size as the unit of sharing and a fixed grain size may be used for all applications. Therefore, it is simple and easy to design such a DSM system. It also allows applications to impose whatever data structures they want on the shared memory. IVY [Li and Hudak 1989] and Mether [Minnich and Farber 1989] use this approach.

2. *Structuring by data type*. In this method, the shared-memory space is structured either as a collection of objects (as in Clouds [Dasgupta et al. 1991] and Orca [Bal et al. 1992]) or as a collection of variables in the source language (as in Munin [Bennett et al. 1990] and Midway [Bershad et al. 1993]). The granularity in such DSM systems is an object or a variable. But since the sizes of the objects and data types vary greatly, these DSM systems use variable grain size to match the size of the object/variable being accessed by the application. The use of variable grain size complicates the design and implementation of these DSM systems.

3. *Structuring as a database*. Another method is to structure the shared memory like a database. For example, Linda [Carriero and Gelernter 1989] takes this approach. Its shared-memory space is ordered as an associative memory (a memory addressed by content rather than by name or address) called a *tuple space*, which is a collection of immutable tuples with typed data items in their fields. A set of primitives that can be added to any base language (such as C and FORTRAN) are provided to place tuples in the tuple space and to read or extract them from tuple space. To perform updates, old data items in the DSM are replaced by new data items. Processes select tuples by specifying the number of their fields and their values or types. Although this structure allows the location of data to be separated from its value, it requires programmers to use special

access functions to interact with the shared-memory space. Therefore, access to shared data is nontransparent. In most other systems, access to shared data is transparent.

5.6 CONSISTENCY MODELS

Consistency requirements vary from application to application [Bennett et al. 1990, Cheriton 1986, Garcia-Molina and Wiederhold 1982]. A *consistency model* basically refers to the degree of consistency that has to be maintained for the shared-memory data for the memory to work correctly for a certain set of applications. It is defined as a set of rules that applications must obey if they want the DSM system to provide the degree of consistency guaranteed by the consistency model. Several consistency models have been proposed in the literature. Of these, the main ones are described below.

It may be noted here that the investigation of new consistency models is currently an active area of research. The basic idea is to invent a consistency model that can allow consistency requirements to be relaxed to a greater degree than existing consistency models, with the relaxation done in such a way that a set of applications can function correctly. This helps in improving the performance of these applications because better concurrency can be achieved by relaxing the consistency requirement. However, applications that depend on a stronger consistency model may not perform correctly if executed in a system that supports only a weaker consistency model. This is because if a system supports the stronger consistency model, then the weaker consistency model is automatically supported but the converse is not true.

Strict Consistency Model

The *strict consistency model* is the strongest form of memory coherence, having the most stringent consistency requirement. A shared-memory system is said to support the strict consistency model if the value returned by a read operation on a memory address is always the same as the value written by the most recent write operation to that address, irrespective of the locations of the processes performing the read and write operations. That is, all writes instantaneously become visible to all processes.

Implementation of the strict consistency model requires the existence of an absolute global time so that memory read/write operations can be correctly ordered to make the meaning of "most recent" clear. However, as explained in Chapter 6, absolute synchronization of clocks of all the nodes of a distributed system is not possible. Therefore, the existence of an absolute global time in a distributed system is also not possible. Consequently, implementation of the strict consistency model for a DSM system is practically impossible.

Sequential Consistency Model

The *sequential consistency model* was proposed by Lamport [1979]. A shared-memory system is said to support the sequential consistency model if all processes see the same order of all memory access operations on the shared memory. The exact order in which the

memory access operations are interleaved does not matter. That is, if the three operations read (r_1), write (w_1), read (r_2) are performed on a memory address in that order, any of the orderings (r_1, w_1, r_2), (r_1, r_2, w_1), (w_1, r_1, r_2), (w_1, r_2, r_1), (r_2, r_1, w_1), (r_2, w_1, r_1) of the three operations is acceptable provided all processes see the same ordering. If one process sees one of the orderings of the three operations and another process sees a different one, the memory is not a sequentially consistent memory. Note here that the only acceptable ordering for a strictly consistent memory is (r_1, w_1, r_2).

The consistency requirement of the sequential consistency model is weaker than that of the strict consistency model because the sequential consistency model does not guarantee that a read operation on a particular memory address always returns the same value as written by the most recent write operation to that address. As a consequence, with a sequentially consistent memory, running a program twice may not give the same result in the absence of explicit synchronization operations. This problem does not exist in a strictly consistent memory.

A DSM system supporting the sequential consistency model can be implemented by ensuring that no memory operation is started until all the previous ones have been completed. A sequentially consistent memory provides *one-copy/single-copy* semantics because all the processes sharing a memory location always see exactly the same contents stored in it. This is the most intuitively expected semantics for memory coherence. Therefore, sequential consistency is acceptable by most applications.

Causal Consistency Model

The *causal consistency model*, proposed by Hutto and Ahamad [1990], relaxes the requirement of the sequential consistency model for better concurrency. Unlike the sequential consistency model, in the causal consistency model, all processes see only those memory reference operations in the same (correct) order that are potentially causally related. Memory reference operations that are not potentially causally related may be seen by different processes in different orders. A memory reference operation (read/write) is said to be potentially causally related to another memory reference operation if the first one might have been influenced in any way by the second one. For example, if a process performs a read operation followed by a write operation, the write operation is potentially causally related to the read operation because the computation of the value written may have depended in some way on the value obtained by the read operation. On the other hand, a write operation performed by one process is not causally related to a write operation performed by another process if the first process has not read either the value written by the second process or any memory variable that was directly or indirectly derived from the value written by the second process.

A shared memory system is said to support the causal consistency model if all write operations that are potentially causally related are seen by all processes in the same (correct) order. Write operations that are not potentially causally related may be seen by different processes in different orders. Note that "correct order" means that if a write operation (w_2) is causally related to another write operation (w_1), the acceptable order is (w_1, w_2) because the value written by w_2 might have been influenced in some way by the value written by w_1. Therefore, (w_2, w_1) is not an acceptable order.

Obviously, in the implementation of a shared-memory system supporting the causal consistency model, there is a need to keep track of which memory reference operation is dependent on which other memory reference operations. This can be done by constructing and maintaining a dependency graph for the memory access operations.

Pipelined Random-Access Memory Consistency Model

The *pipelined random-access memory (PRAM) consistency model*, proposed by Lipton and Sandberg [1988], provides a weaker consistency semantics than the consistency models described so far. It only ensures that all write operations performed by a single process are seen by all other processes in the order in which they were performed as if all the write operations performed by a single process are in a pipeline. Write operations performed by different processes may be seen by different processes in different orders. For example, if w_{11} and w_{12} are two write operations performed by a process P_1 in that order, and w_{21} and w_{22} are two write operations performed by a process P_2 in that order, a process P_3 may see them in the order $[(w_{11}, w_{12}), (w_{21}, w_{22})]$ and another process P_4 may see them in the order $[(w_{21}, w_{22}), (w_{11}, w_{12})]$.

The PRAM consistency model is simple and easy to implement and also has good performance. It can be implemented by simply sequencing the write operations performed at each node independently of the write operations performed on other nodes. It leads to better performance than the previous models because a process need not wait for a write operation performed by it to complete before starting the next one since all write operations of a single process are pipelined. Notice that in sequential consistency all processes agree on the same order of memory reference operations, but in PRAM consistency all processes do not agree on the same order of memory reference operations. Therefore, for the example given above, either $[(w_{11}, w_{12}), (w_{21}, w_{22})]$ or $[(w_{21}, w_{22}), (w_{11}, w_{12})]$ is an acceptable ordering for sequential consistency but not both. That is, unlike PRAM consistency, both processes P_3 and P_4 must agree on the same order.

Processor Consistency Model

The *processor consistency model*, proposed by Goodman [1989], is very similar to the PRAM consistency model with an additional restriction of memory coherence. That is, a processor consistent memory is both coherent and adheres to the PRAM consistency model. Memory coherence means that for any memory location all processes agree on the same order of all write operations to that location. In effect, processor consistency ensures that all write operations performed on the same memory location (no matter by which process they are performed) are seen by all processes in the same order. This requirement is in addition to the requirement imposed by the PRAM consistency model. Therefore, in the example given for PRAM consistency, if w_{12} and w_{22} are write operations for writing to the same memory location x, all processes must see them in the same order—w_{12} before w_{22} or w_{22} before w_{12}. This means that for processor consistency both processes P_3 and P_4 must see the write operations in the same order, which may be either $[(w_{11}, w_{12}), (w_{21}, w_{22})]$ or $[(w_{21}, w_{22}), (w_{11}, w_{12})]$. Notice that, for this example, processor consistency and

sequential consistency lead to the same final result, but this may not be true for other cases.

Weak Consistency Model

The *weak consistency model*, proposed by Dubois et al. [1988], is designed to take advantage of the following two characteristics common to many applications:

1. It is not necessary to show the change in memory done by every write operation to other processes. The results of several write operations can be combined and sent to other processes only when they need it. For example, when a process executes in a critical section, other processes are not supposed to see the changes made by the process to the memory until the process exits from the critical section. In this case, all changes made to the memory by the process while it is in its critical section need be made visible to other processes only at the time when the process exits from the critical section.

2. Isolated accesses to shared variables are rare. That is, in many applications, a process makes several accesses to a set of shared variables and then no access at all to the variables in this set for a long time.

Both characteristics imply that better performance can be achieved if consistency is enforced on a group of memory reference operations rather than on individual memory reference operations. This is exactly the basic idea behind the weak consistency model.

The main problem in implementing this idea is determining how the system can know that it is time to show the changes performed by a process to other processes since this time is different for different applications. Since there is no way for the system to know this on its own, the programmers are asked to tell this to the system for their applications. For this, a DSM system that supports the weak consistency model uses a special variable called a *synchronization variable*. The operations on it are used to synchronize memory. That is, when a synchronization variable is accessed by a process, the entire (shared) memory is synchronized by making all the changes to the memory made by all processes visible to all other processes. Note that memory synchronization in a DSM system will involve propagating memory updates done at a node to all other nodes having a copy of the same memory addresses.

For supporting weak consistency, the following requirements must be met:

1. All accesses to synchronization variables must obey sequential consistency semantics.
2. All previous write operations must be completed everywhere before an access to a synchronization variable is allowed.
3. All previous accesses to synchronization variables must be completed before access to a nonsynchronization variable is allowed.

Note that the weak consistency model provides better performance at the cost of putting extra burden on the programmers.

Release Consistency Model

We saw that in the weak consistency model the entire (shared) memory is synchronized when a synchronization variable is accessed by a process, and memory synchronization basically involves the following operations:

1. All changes made to the memory by the process are propagated to other nodes.
2. All changes made to the memory by other processes are propagated from other nodes to the process's node.

A closer observation shows that this is not really necessary because the first operation need only be performed when the process exits from a critical section and the second operation need only be performed when the process enters a critical section. Since a single synchronization variable is used in the weak consistency model, the system cannot know whether a process accessing a synchronization variable is entering a critical section or exiting from a critical section. Therefore, both the first and second operations are performed on every access to a synchronization variable by a process. For better performance, the *release consistency model* [Gharachorloo et al. 1990] provides a mechanism to clearly tell the system whether a process is entering a critical section or exiting from a critical section so that the system can decide and perform only either the first or the second operation when a synchronization variable is accessed by a process. This is achieved by using two synchronization variables (called *acquire* and *release*) instead of a single synchronization variable. *Acquire* is used by a process to tell the system that it is about to enter a critical section, so that the system performs only the second operation when this variable is accessed. On the other hand, *release* is used by a process to tell the system that it has just exited a critical section, so that the system performs only the first operation when this variable is accessed. Programmers are responsible for putting *acquire* and *release* at suitable places in their programs.

Release consistency may also be realized by using the synchronization mechanism based on barriers instead of critical sections. A *barrier* defines the end of a phase of execution of a group of concurrently executing processes. All processes in the group must complete their execution up to a barrier before any process is allowed to proceed with its execution following the barrier. That is, when a process of a group encounters a barrier during its execution, it blocks until all other processes in the group complete their executions up to the barrier. When the last process completes its execution up to the barrier, all shared variables are synchronized and then all processes resume with their executions. Therefore, *acquire* is departure from a barrier, and *release* is completion of the execution of the last process up to a barrier.

A barrier can be implemented by using a centralized barrier server. When a barrier is created, it is given a count of the number of processes that must be waiting on it before they can all be released. Each process of a group of concurrently executing processes sends a message to the barrier server when it arrives at a barrier and then blocks until a reply is received from the barrier server. The barrier server does not send any replies until all processes in the group have arrived at the barrier.

For supporting release consistency, the following requirements must be met:

1. All accesses to *acquire* and *release* synchronization variables obey processor consistency semantics.
2. All previous *acquires* performed by a process must be completed successfully before the process is allowed to perform a data access operation on the memory.
3. All previous data access operations performed by a process must be completed successfully before a *release* access done by the process is allowed.

Note that *acquires* and *releases* of locks of different critical regions (or barriers) can occur independently of each other. Gharachorloo et al. [1990] showed that if processes use appropriate synchronization accesses properly, a release consistent DSM system will produce the same results for an application as that if the application was executed on a sequentially consistent DSM system.

A variation of release consistency is *lazy release consistency*, proposed by Keleher et al. [1992], and is more efficient than the conventional release consistency. In the conventional approach, when a process does a *release* access, the contents of all the modified memory locations at the process's node are sent to all other nodes that have a copy of the same memory locations in their local cache. However, in the lazy approach, the modifications are not sent to other nodes at the time of release. Rather, these modifications are sent to other nodes only on demand. That is, when a process does an *acquire* access, all modifications of other nodes are acquired by the process's node. Therefore, the modifications that were to be sent to this node at the time of *release* access will be received by the node now (exactly when it is needed there). Lazy release consistency has better performance because in this method no network traffic is generated at all until an *acquire* access is done by a process at some other node.

Discussion of Consistency Models

In the description above, we saw some of the main consistency models. Several others have been proposed in the literature. A nice overview of the consistency models can be found in [Mosberger 1993]. It is difficult to grade the consistency models based on performance because quite different results are usually obtained for different applications. That is, one application may have good performance for one model, but another application may have good performance for some other model. Therefore, in the design of a DSM system, the choice of a consistency model usually depends on several other factors, such as how easy is it to implement, how close is its semantics to intuition, how much concurrency does it allow, and how easy is it to use (does it impose extra burden on the programmers).

Among the consistency models described above, strict consistency is never used in the design of a DSM system because its implementation is practically impossible. The most commonly used model in DSM systems is the sequential consistency model because it can be implemented, it supports the most intuitively expected semantics for memory coherence, and it does not impose any extra burden on the programmers. Another

important reason for its popularity is that a sequentially consistent DSM system allows existing multiprocessor programs to be run on multicomputer architectures without modification. This is because programs written for multiprocessors normally assume that memory is sequentially consistent. However, it is very restrictive and hence suffers from the drawback of low concurrency. Therefore, several DSM systems are designed to use other consistency models that are weaker than sequential consistency.

Causal consistency, PRAM consistency, processor consistency, weak consistency, and release consistency are the main choices in the weaker category. The main problem with the use of causal consistency, PRAM consistency, and processor consistency models in the design of a DSM system is that they do not support the intuitively expected semantics for memory coherence because different processes may see different sequences of operations. Therefore, with these three models it becomes the responsibility of the programmers to avoid doing things that work only if the memory is sequentially consistent. In this respect, these models impose extra burden on the programmers.

Weak consistency and release consistency models, which use explicit synchronization variables, seem more promising for use in DSM design because they provide better concurrency and also support the intuitively expected semantics. The only problem with these consistency models is that they require the programmers to use the synchronization variables properly. This imposes some burden on the programmers.

Note that one of the main reasons for taking all the trouble to implement a DSM system is to support the shared-memory paradigm to make programming of distributed applications simpler than in the message-passing paradigm. Therefore, some DSM system designers are of the opinion that no extra burden should be imposed on the programmers. Designers having this view prefer to use the sequential consistency model.

5.6.1 Implementing Sequential Consistency Model

We saw above that the most commonly used consistency model in DSM systems is the sequential consistency model. Hence, a description of the commonly used protocols for implementing sequentially consistent DSM systems is presented below. A protocol for implementing a release-consistent DSM system will be presented in the next section.

Protocols for implementing the sequential consistency model in a DSM system depend to a great extent on whether the DSM system allows replication and/or migration of shared-memory data blocks. The designer of a DSM system may choose from among the following replication and migration strategies [Stumm and Zhou 1990]:

1. Nonreplicated, nonmigrating blocks (NRNMBs)
2. Nonreplicated, migrating blocks (NRMBs)
3. Replicated, migrating blocks (RMBs)
4. Replicated, nonmigrating blocks (RNMBs)

The protocols that may be used for each of these categories are described below. The data-locating mechanisms suitable for each category have also been described. Several of the ideas presented below were first proposed and used in the IVY system [Li 1988].

Nonreplicated, Nonmigrating Blocks

This is the simplest strategy for implementing a sequentially consistent DSM system. In this strategy, each block of the shared memory has a single copy whose location is always fixed. All access requests to a block from any node are sent to the *owner node* of the block, which has the only copy of the block. On receiving a request from a client node, the memory management unit (MMU) and operating system software of the owner node perform the access request on the block and return a response to the client (Fig. 5.3).

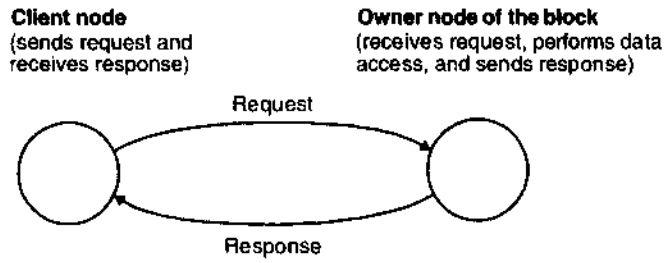

Fig. 5.3 Nonreplicated, nonmigrating blocks (NRNMB) strategy.

Enforcing sequential consistency is trivial in this case because a node having a shared block can merely perform all the access requests (on the block) in the order it receives them.

Although the method is simple and easy to implement, it suffers from the following drawbacks:

- Serializing data access creates a bottleneck.
- Parallelism, which is a major advantage of DSM, is not possible with this method.

Data Locating in the NRNMB Strategy. The NRNMB strategy has the following characteristics:

1. There is a single copy of each block in the entire system.
2. The location of a block never changes.

Based on these characteristics, the best approach for locating a block in this case is to use a simple mapping function to map a block to a node. When a fault occurs, the fault handler of the faulting node uses the mapping function to get the location of the accessed block and forwards the access request to that node.

Nonreplicated, Migrating Blocks

In this strategy each block of the shared memory has a single copy in the entire system. However, each access to a block causes the block to migrate from its current node to the node from where it is accessed. Therefore, unlike the previous strategy in which the owner node of a block always remains fixed, in this strategy the owner node of a block changes as soon as the block is migrated to a new node (Fig. 5.4). When a block is migrated away, it is removed from any local address space it has been mapped into. Notice that in this strategy only the processes executing on one node can read or write a given data item at any one time. Therefore the method ensures sequential consistency.

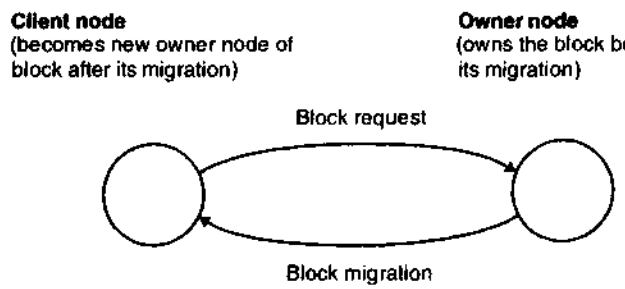

Fig. 5.4 Nonreplicated, migrating blocks (NRMB) strategy.

The method has the following advantages [Stumm and Zhou 1990]:

1. No communication costs are incurred when a process accesses data currently held locally.
2. It allows the applications to take advantage of data access locality. If an application exhibits high locality of reference, the cost of data migration is amortized over multiple accesses.

However, the method suffers from the following drawbacks:

1. It is prone to thrashing problem. That is, a block may keep migrating frequently from one node to another, resulting in few memory accesses between migrations and thereby poor performance.
2. The advantage of parallelism cannot be availed in this method also.

Data Locating in the NRMB Strategy. In the NRMB strategy, although there is a single copy of each block, the location of a block keeps changing dynamically. Therefore, one of the following methods may be used in this strategy to locate a block:

1. *Broadcasting.* In this method, each node maintains an *owned blocks table* that contains an entry for each block for which the node is the current owner (Fig. 5.5). When

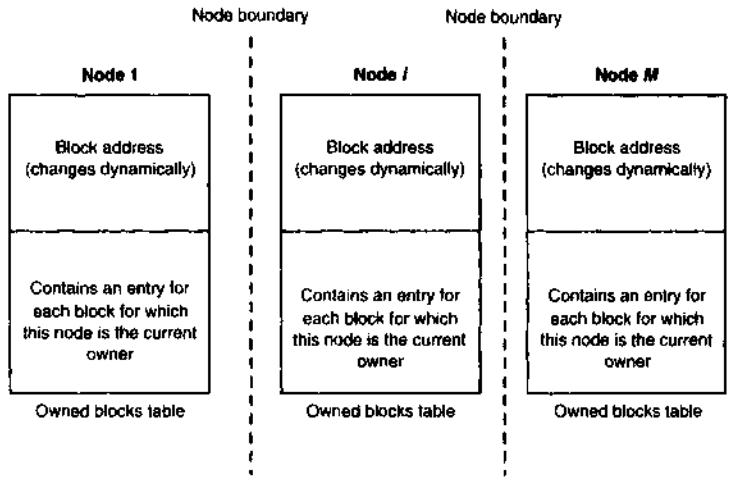

Fig. 5.5 Structure and locations of owned blocks table in the broadcasting data-locating mechanism for NRMB strategy.

a fault occurs, the fault handler of the faulting node broadcasts a read/write request on the network. The node currently having the requested block then responds to the broadcast request by sending the block to the requesting node.

A major disadvantage of the broadcasting algorithm is that it does not scale well. When a request is broadcast, not just the node that has the requested block but all nodes must process the broadcast request. This makes the communication subsystem a potential bottleneck. The network latency of a broadcast may also require accesses to take a long time to complete.

2. *Centralized-server algorithm.* In this method, a centralized server maintains a block table that contains the location information for all blocks in the shared-memory space (Fig. 5.6). The location and identity of the centralized server is well known to all nodes.

When a fault occurs, the fault handler of the faulting node (N) sends a request for the accessed block to the centralized server. The centralized server extracts the location information of the requested block from the block table, forwards the request to that node, and changes the location information in the corresponding entry of the block table to node N. On receiving the request, the current owner transfers the block to node N, which becomes the new owner of the block.

The centralized-server method suffers from two drawbacks: (a) the centralized server serializes location queries, reducing parallelism, and (b) the failure of the centralized server will cause the DSM system to stop functioning.

3. *Fixed distributed-server algorithm.* The fixed distributed-server scheme is a direct extension of the centralized-server scheme. It overcomes the problems of the centralized-server scheme by distributing the role of the centralized server. Therefore, in this scheme, there is a block manager on several nodes, and each block manager is given a

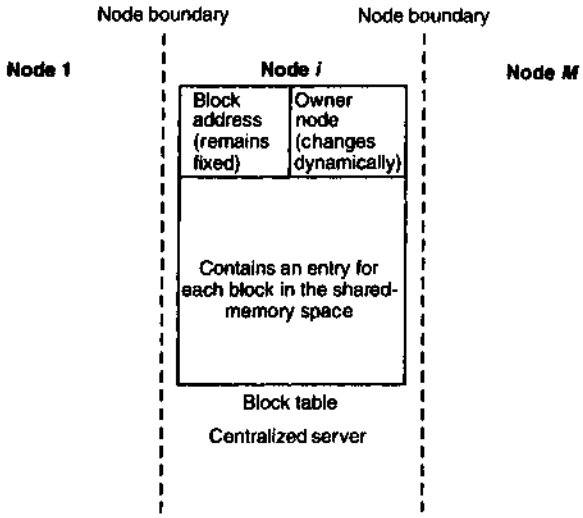

Fig. 5.6 Structure and location of block table in the centralized-server data-locating mechanism for NRMB strategy.

predetermined subset of data blocks to manage (Fig. 5.7). The mapping from data blocks to block managers and their corresponding nodes is described by a mapping function. Whenever a fault occurs, the mapping function is used by the fault handler of the faulting node to find out the node whose block manager is managing the currently accessed block. Then a request for the block is sent to the block manager of that node. The block manager handles the request exactly in the same manner as that described for the centralized-server algorithm.

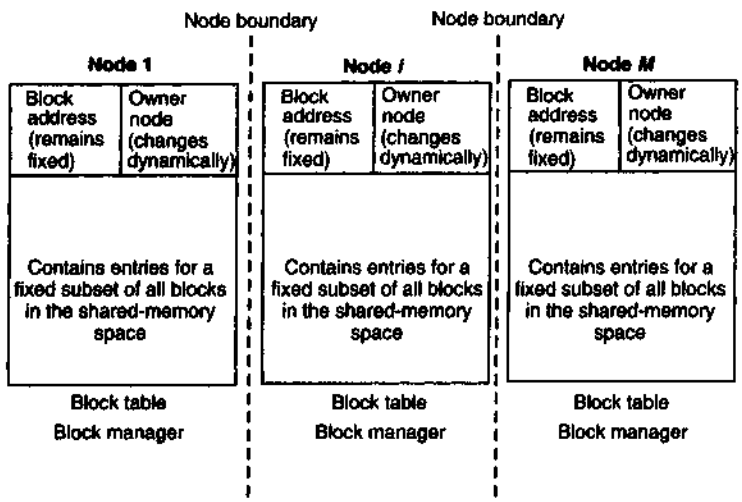

Fig. 5.7 Structure and locations of block table in the fixed distributed-server data-locating mechanism for NRMB strategy.

4. *Dynamic distributed-server algorithm.* This scheme does not use any block manager and attempts to keep track of the ownership information of all blocks in each node. For this, each node has a block table that contains the ownership information for all blocks in the shared-memory space (Fig. 5.8). However, the information contained in the ownership field is not necessarily correct at all times, but if incorrect, it at least provides the beginning of a sequence of nodes to be traversed to reach the true owner node of a block. Therefore, this field gives the node a hint on the location of the owner of a block and hence is called the *probable owner.*

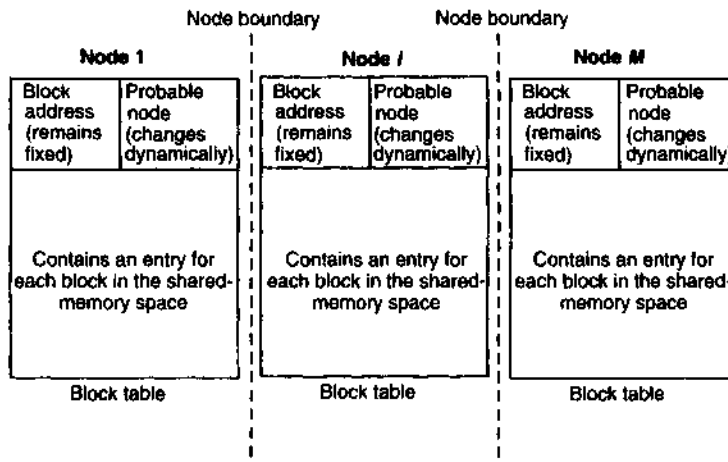

Fig. 5.8 Structure and locations of block table in the dynamic distributed-server data-locating mechanism for NRMB strategy.

When a fault occurs, the faulting node (N) extracts from its local block table the node information stored in the probable owner field of the entry for the accessed block. It then sends a request for the block to that node. If that node is the true owner of the block, it transfers the block to node N and updates the location information of the block in its local block table to node N. Otherwise, it looks up its local block table, forwards the request to the node indicated in the probable-owner field of the entry for the block, and updates the value of this field to node N. When node N receives the block, it becomes the new owner of the block.

Replicated, Migrating Blocks

A major disadvantage of the nonreplication strategies is lack of parallelism because only the processes on one node can access data contained in a block at any given time. To increase parallelism, virtually all DSM systems replicate blocks. With replicated blocks, read operations can be carried out in parallel at multiple nodes by accessing the local copy of the data. Therefore, the average cost of read operations is reduced because no

communication overhead is involved if a replica of the data exists at the local node. However, replication tends to increase the cost of write operations because for a write to a block all its replicas must be invalidated or updated to maintain consistency. Nevertheless, if the read/write ratio is large, the extra expense for the write operations may be more than offset by the lower average cost of the read operations.

Replication complicates the memory coherence protocol due to the requirement of keeping multiple copies of a block consistent. The two basic protocols that may be used for ensuring sequential consistency in this case are as follows:

1. *Write-invalidate.* In this scheme, all copies of a piece of data except one are invalidated before a write can be performed on it. Therefore, when a write fault occurs at a node, its fault handler copies the accessed block from one of the block's current nodes to its own node, invalidates all other copies of the block by sending an invalidate message containing the block address to the nodes having a copy of the block, changes the access of the local copy of the block to write, and returns to the faulting instruction (Fig. 5.9). After returning, the node "owns" that block and can proceed with the write operation and other read/write operations until the block ownership is relinquished to some other node. Notice that in this method, after invalidation of a block, only the node that performs the write operation on the block holds the modified version of the block.

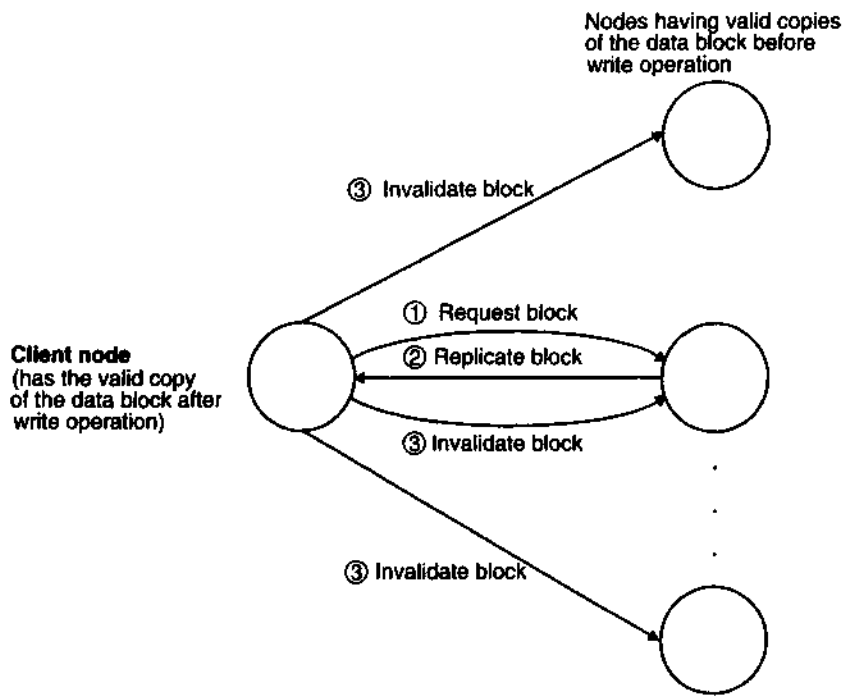

Fig. 5.9 Write-invalidate memory coherence approach for replicated, migrating blocks (RMB) strategy.

If one of the nodes that had a copy of the block before invalidation tries to perform a memory access operation (read/write) on the block after invalidation, a cache miss will occur and the fault handler of that node will have to fetch the block again from a node having a valid copy of the block. Therefore the scheme achieves sequential consistency.

2. *Write-update.* In this scheme, a write operation is carried out by updating all copies of the data on which the write is performed. Therefore, when a write fault occurs at a node, the fault handler copies the accessed block from one of the block's current nodes to its own node, updates all copies of the block by performing the write operation on the local copy of the block and sending the address of the modified memory location and its new value to the nodes having a copy of the block, and then returns to the faulting instruction (Fig. 5.10). The write operation completes only after all the copies of the block have been successfully updated. Notice that in this method, after a write operation completes, all the nodes that had a copy of the block before the write also have a valid copy of the block after the write. In this method, sequential consistency can be achieved by using a mechanism to totally order the write operations of all the nodes so that all processes agree on the order of writes. One method to do this is to use a global sequencer to sequence the write operations of all nodes. In this method, the intended modification of each write operation is first sent to the global sequencer. The sequencer assigns the next

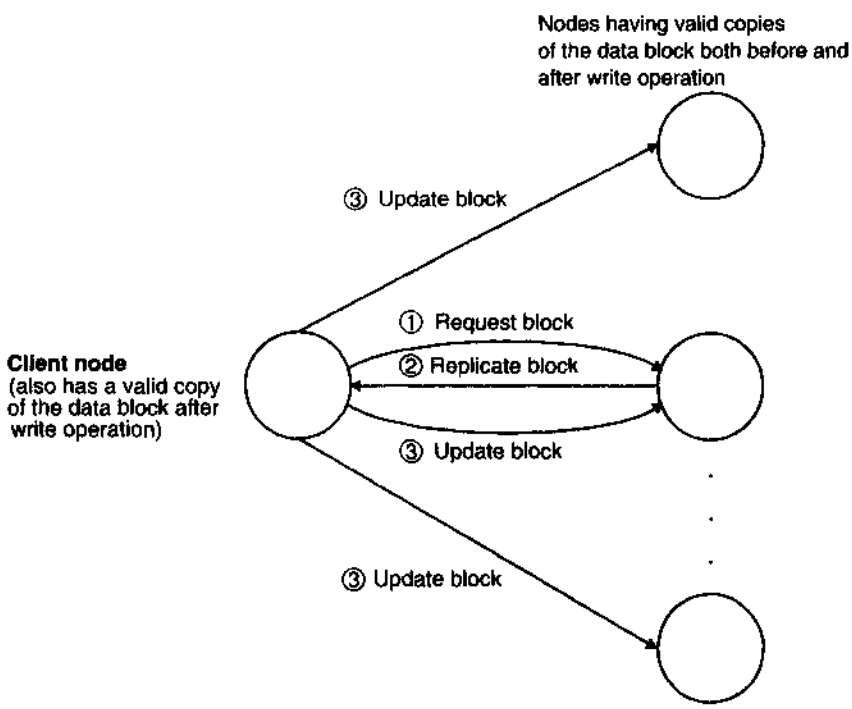

Fig. 5.10 Write-update memory coherence approach for replicated, migrating blocks (RMB) strategy.

sequence number to the modification and multicasts the modification with this sequence number to all the nodes where a replica of the data block to be modified is located (Fig. 5.11). The write operations are processed at each node in sequence number order. That is, when a new modification arrives at a node, its sequence number is verified as the next expected one. If the verification fails, either a modification was missed or a modification was received out of order, in which case the node requests the sequencer for a retransmission of the missing modification. Obviously, a log of recent write requests must be maintained somewhere in this method. Note that the set of reads that take place between any two consecutive writes is well defined, and their ordering is immaterial to sequential consistency.

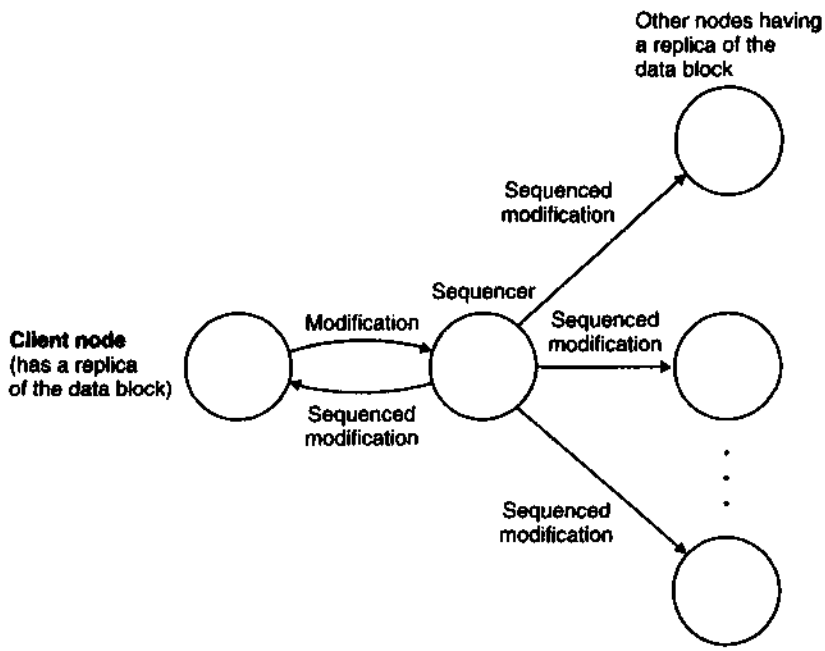

Fig. 5.11 Global sequencing mechanism to sequence the write operations of all nodes.

The write-update approach is very expensive for use with loosely coupled distributed-memory systems because it requires a network access on every write operation and updates all copies of the modified block. On the other hand, in the write-invalidate approach, updates are only propagated when data are read, and several updates can take place (because many programs exhibit locality of reference) before communication is necessary. Therefore, most DSM systems use the write-invalidate protocol. In the basic implementation approach of the write-invalidate protocol, there is a status tag associated with each block. The status tag indicates whether the block is valid, whether it is shared, and whether it is read-only or writable. With this status information, read and write requests are carried out in the following manner [Nitzberg and Virginia Lo 1991]:

Read Request

1. If there is a local block containing the data and if it is valid, the request is satisfied by accessing the local copy of the data.
2. Otherwise, the fault handler of the requesting node generates a read fault and obtains a copy of the block from a node having a valid copy of the block. If the block was writable on another node, this read request will cause it to become read-only. The read request is now satisfied by accessing the data from the local block, which remains valid until an invalidate request is received.

Write Request

1. If there is a local block containing the data and if it is valid and writable, the request is immediately satisfied by accessing the local copy of the data.
2. Otherwise, the fault handler of the requesting node generates a write fault and obtains a valid copy of the block and changes its status to writable. A write fault for a block causes the invalidation of all other copies of the block. When the invalidation of all other copies of the block completes, the block is valid locally and writable, and the original write request may now be performed.

Data Locating in the RMB Strategy. The following data-locating issues are involved in the write-invalidate protocol used with the RMB strategy:

1. Locating the owner of a block. An *owner* of a block is the node that owns the block, namely, the most recent node to have write access to it.
2. Keeping track of the nodes that currently have a valid copy of the block.

One of the following algorithms may be used to address these two issues:

1. *Broadcasting.* In this method, each node has an owned blocks table (Fig. 5.12). This table of a node has an entry for each block for which the node is the owner. Each entry of this table has a *copy-set* field that contains a list of nodes that currently have a valid copy of the corresponding block.

When a read fault occurs, the faulting node (N) sends a broadcast read request on the network to find the owner of the required block. The owner of the block responds by adding node N to the block's copy-set field in its owned blocks table and sending a copy of the block to node N. Similarly, when a write fault occurs, the faulting node sends a broadcast write request on the network for the required block. On receiving this request, the owner of the block relinquishes its ownership to node N and sends the block and its copy set to node N. When node N receives the block and the copy set, it sends an invalidation message to all nodes in the copy set. Node N now becomes the new owner of the block, and therefore an entry is made for the block in its local-owned blocks table. The copy-set field of the entry is initialized to indicate that there are no other copies of the block since all the copies were invalidated.

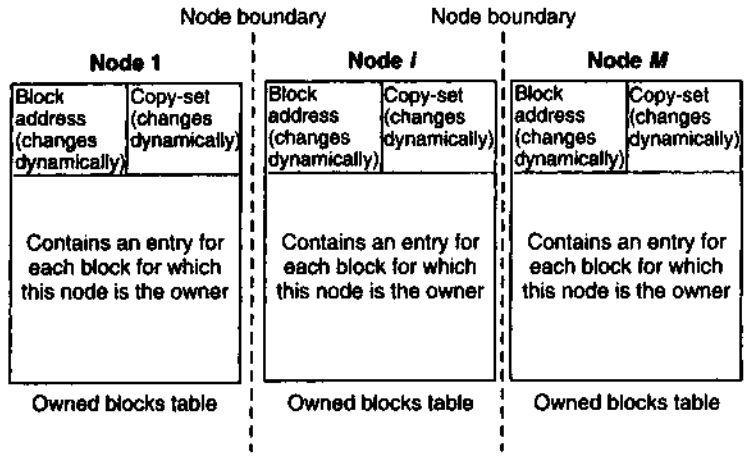

Fig. 5.12 Structure and locations of owned blocks table in the broadcasting data-locating mechanism for RMB strategy.

The method of broadcasting suffers from the disadvantages mentioned during the description of the data-locating mechanisms for the NRMB strategy.

2. *Centralized-server algorithm.* This method is similar to the centralized-server algorithm of the NRMB strategy. However, in this case, each entry of the block table, managed by the centralized server, has an *owner-node* field that indicates the current owner node of the block and a *copy-set* field that contains a list of nodes having a valid copy of the block (Fig. 5.13).

When a read/write fault occurs, the faulting node (N) sends a read/write fault request for the accessed block to the centralized server. For a read fault, the centralized server adds node N to the block's copy set and returns the owner node information to node N. On the other hand, for a write fault, it returns both the copy set and owner node information to node N and then initializes the copy-set field to contain only node N. Node N then sends a request for the block to the owner node. On receiving this request, the owner node returns a copy of the block to node N. In a write fault, node N also sends an invalidate message to all nodes in the copy set. Node N can then perform the read/write operation.

The disadvantages of the centralized-server algorithm have already been mentioned during description of the data-locating mechanisms for the NRMB strategy.

3. *Fixed distributed-server algorithm.* This scheme is a direct extension of the centralized-server scheme. In this scheme, the role of the centralized server is distributed to several distributed servers. Therefore, in this scheme, there is a block manager on several nodes (Fig. 5.14). Each block manager manages a predetermined subset of blocks, and a mapping function is used to map a block to a particular block manager and its corresponding node. When a fault occurs, the mapping function is used to find the location of the block manager that is managing the currently requested block. Then a request for

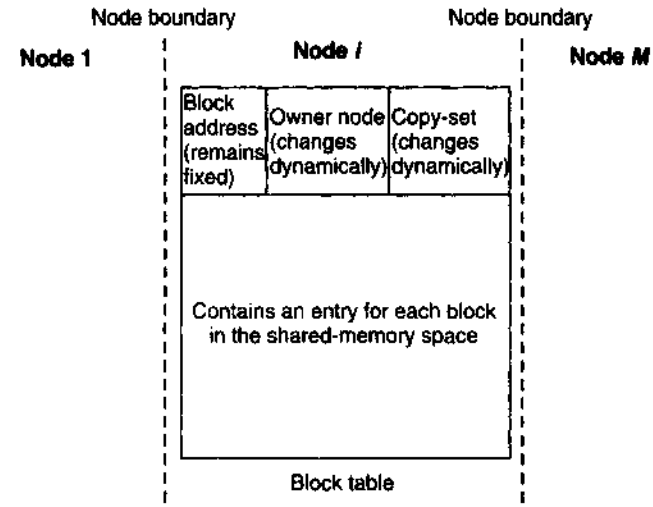

Fig. 5.13 Structure and location of block table in the centralized-server data-locating mechanism for RMB strategy.

the accessed block is sent to the block manager of that node. A request is handled in exactly the same manner as that described for the centralized-server algorithm.

The advantages and limitations of the fixed distributed-server algorithm have already been mentioned during the description of the data-locating mechanisms for the NRMB strategy.

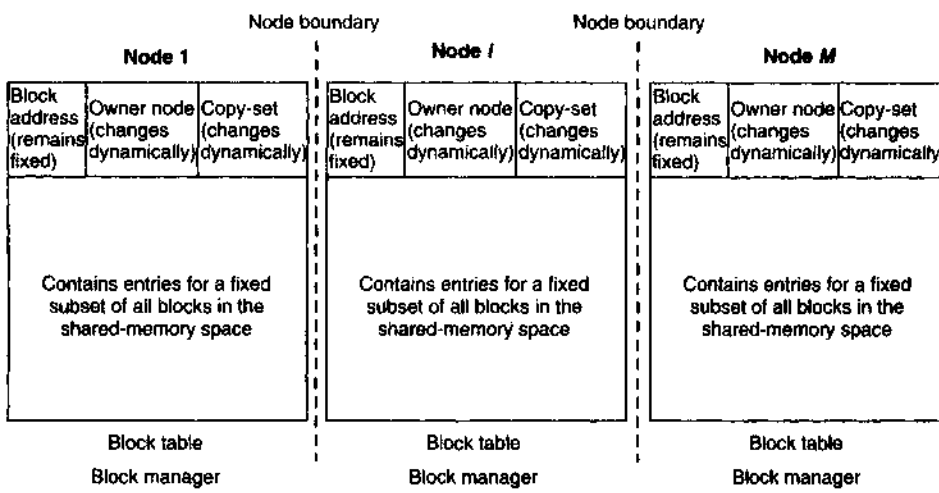

Fig. 5.14 Structure and locations of block table in the fixed distributed-server data-locating mechanism for RMB strategy.

4. *Dynamic distributed-server algorithm.* This scheme works in a similar manner as the dynamic distributed-server algorithm of the NRMB strategy. Each node has a block table that contains an entry for all blocks in the shared-memory space (Fig. 5.15). Each entry of the table has a probable-owner field that gives the node a hint on the location of the owner of the corresponding block. In addition, if a node is the true owner of a block, the entry for the block in the block table of the node also contains a copy-set field that provides a list of nodes having a valid copy of the block.

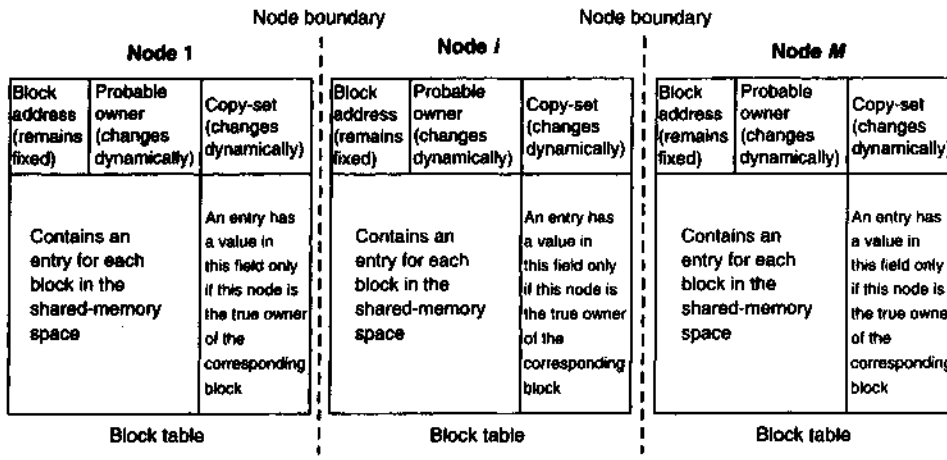

Fig. 5.15 Structure and locations of block table in the dynamic distributed-server data-locating mechanism for RMB strategy.

When a fault occurs, the fault handler of the faulting node (N) extracts the probable-owner node information for the accessed block from its local block table and sends a request for the block to that node. If that node is not the true owner of the block, it looks up its local block table and forwards the request to the node indicated in the probable-owner field of the entry for the block. On the other hand, if the node is the true owner of the block, it proceeds with the request as follows. For a read fault, it adds node N in the copy-set field of the entry corresponding to the block and sends a copy of the block to node N, which then performs the read operation. For a write fault, it sends a copy of the block and its copy-set information to node N and deletes the copy-set information of that block from the local block table. On receiving the block and copy-set information, node N sends an invalidation request to all nodes in the copy set. After this, it becomes the new owner of the block, updates its local block table, and proceeds with performing the write operation.

To reduce the length of the chain of nodes to be traversed to reach the true owner of a block, the probable-owner field of a block in a node's block table is updated as follows [Li and Hudak 1989]:

(a) Whenever the node receives an invalidation request

(b) Whenever the node relinquishes ownership, that is, on a write fault

(c) Whenever the node forwards a fault request

In the first two cases, the probable-owner field is changed to the new owner of the block. In the third case, the probable-owner field is changed to the original faulting node (N). This is because, if the request is for write, the faulting node (N) is going to be the new owner. If the request is for read, we know that after the request is satisfied, the faulting node (N) will have the correct ownership information. In either case, it is a good idea to change the probable-owner field of the block to node N.

It has been proved in [Li and Hudak 1989] that a fault on any node for a block eventually reaches the true owner of the block, and if there are altogether M nodes, it will take at most $M - 1$ messages to locate the true owner of a block.

Some refinements of the data-locating algorithms described above may be found in [Li and Hudak 1989].

Replicated, Nonmigrating Blocks

In this strategy, a shared-memory block may be replicated at multiple nodes of the system, but the location of each replica is fixed. A read or write access to a memory address is carried out by sending the access request to one of the nodes having a replica of the block containing the memory address. All replicas of a block are kept consistent by updating them all in case of a write access. A protocol similar to the write-update protocol is used for this purpose. Sequential consistency is ensured by using a global sequencer to sequence the write operations of all nodes (Fig. 5.11).

Data Locating in the RNMB Strategy. The RNMB strategy has the following characteristics:

1. The replica locations of a block never change.
2. All replicas of a data block are kept consistent.
3. Only a read request can be directly sent to one of the nodes having a replica of the block containing the memory address on which the read request is performed and all write requests have to be first sent to the sequencer.

Based on these characteristics, the best approach of data locating for handling read/write operations in this case is to have a block table at each node and a sequence table with the sequencer (Fig. 5.16). The block table of a node has an entry for each block in the shared memory. Each entry maps a block to one of its replica locations. The sequence table also has an entry for each block in the shared-memory space. Each entry of the sequence table has three fields—a field containing the block address, a replica-

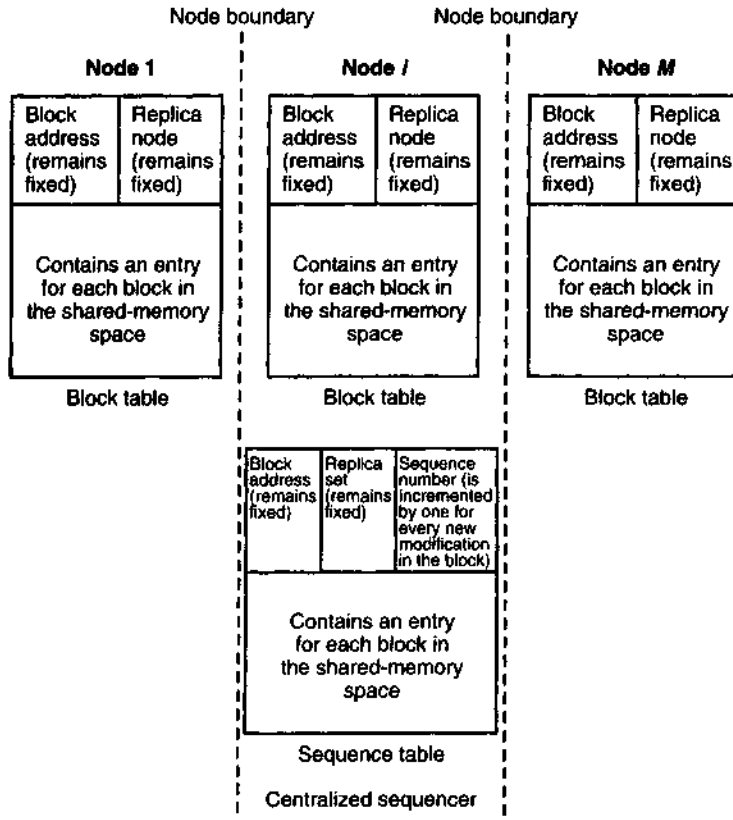

Fig. 5.16 Structure and locations of block table and sequence table in the centralized sequencer data-locating mechanism for RNMB strategy.

set field containing a list of nodes having a replica of the block, and a sequence number field that is incremented by 1 for every new modification performed on the block.

For performing a read operation on a block, the replica location of the block is extracted from the local block table and the read request is directly sent to that node. A write operation on a block is sent to the sequencer. The sequencer assigns the next sequence number to the requested modification. It then multicasts the modification with this sequence number to all the nodes listed in the replica-set field of the entry for the block. The write operations are performed at each node in sequence number order.

Note that, to prevent all read operations on a block getting serviced at the same replica node, as far as practicable, the block table of different nodes should have different replica locations entered in the entry corresponding to the block. This will help in evenly distributing the read operations on the same block emerging from different nodes.

5.6.2 Munin: A Release Consistent DSM System

We saw in the discussion of consistency models that in addition to sequential consistency, release consistency is also promising and attractive for use in DSM systems. Therefore, as a case study of a release consistent DSM system, a description of the Munin system is presented below [Bennett et al. 1990, Carter et al. 1991, Carter et al. 1994].

Structure of Shared-Memory Space

The shared-memory space of Munin is structured as a collection of *shared variables* (includes program data structures). The shared variables are declared with the keyword *shared* so the compiler can recognize them. A programmer can annotate a shared variable with one of the standard annotation types (annotation types for shared variables are described later). Each shared variable, by default, is placed by the compiler on a separate page that is the unit of data transfer across the network by the MMU hardware. However, programmers can specify that multiple shared variables having the same annotation type be placed in the same page. Placing of variables of different annotation types in the same page is not allowed because the consistency protocol used for a page depends on the annotation type of variables contained in the page. Obviously, variables of size larger than the size of a page occupy multiple pages. The shared variables are declared with the keyword *shared* so the compiler can recognize them.

Implementation of Release Consistency

In the description of release consistency, we saw that for release consistency applications must be modeled around critical sections. Therefore a DSM system that supports release consistency must have mechanisms and programming language constructs for critical sections. Munin provides two such synchronization mechanisms—a locking mechanism and a barrier mechanism.

The locking mechanism uses *lock* synchronization variables with *acquireLock* and *releaseLock* primitives for accessing these variables. The *acquireLock* primitive with a lock variable as its parameter is executed by a process to enter a critical section, and the *releaseLock* primitive with the same lock variable as its parameter is executed by the process to exit from the critical section. To ensure release consistency, write operations on shared variables must be performed only within critical sections, but read operations on shared variables may be performed either within or outside a critical section. Modifications made to a shared variable within a critical section are sent to other nodes having a replica of the shared variable only when the process making the update exits from the critical section. If programs of an application are properly structured as described above, the DSM system will appear to be sequentially consistent.

When a process makes an *acquireLock* request for acquiring a lock variable, the system first checks if the lock variable is available on the local node. If not, the probable-owner mechanism is used to find the location of the current owner of the lock variable, and the request is sent to that node. Whether the lock is on the local or remote node, if it is free, it is granted to the requesting process. Otherwise, the requesting process is added to the end of

the queue of processes waiting to acquire the lock variable. When the lock variable is released by its current owner, it is given to the next process in the waiting queue.

The barrier mechanism uses *barrier* synchronization variables with a *waitAtBarrier* primitive for accessing these variables. Barriers are implemented by using the centralized barrier server mechanism.

In a network page fault, the probable-owner-based dynamic distributed-server algorithm (already described during the description of RMB strategy) is used in Munin to locate a page containing the accessed shared variable. A mechanism similar to the copy-set mechanism (also described during the description of RMB strategy) is used to keep track of all the replica locations of a page.

Annotations for Shared Variables

The release consistency of Munin allows applications to have better performance than in a sequentially consistent DSM system. For further performance improvement, Munin defines several standard annotations for shared variables and uses a different consistency protocol for each type that is most suitable for that type. That is, consistency protocols in this approach are applied at the granularity of individual data items. The standard annotations and the consistency protocol for variables of each type are as follows [Bennett et al. 1990]:

1. *Read-only.* Shared-data variables annotated as read-only are immutable data items. These variables are read but never written after initialization. Therefore, the question of consistency control does not arise. As these variables are never modified, their average read cost can be reduced drastically by freely replicating them on all nodes from where they are accessed. Therefore, when a reference to such a variable causes a network page fault, the page having the variable is copied to the faulting node from one of the nodes already having a copy of the page. Read-only variables are protected by the MMU hardware, and an attempt to write to such a variable causes a fatal error.

2. *Migratory.* Shared variables that are accessed in phases, where each phase corresponds to a series of accesses by a single process, may be annotated as migratory variables. The locking mechanism is used to keep migratory variables consistent. That is, migratory variables are protected by lock synchronization variables and are used within critical sections.

To access a migratory variable, a process first acquires a lock for the variable, uses the variable for some time, and then releases the lock when it has finished using it. At a time, the system allows only a single process to acquire a lock for a migratory variable. If a network page fault occurs when a process attempts to acquire a lock for a migratory variable, the page is migrated to the faulting node from the node that is its current owner. The NRMB strategy is used in this case. That is, pages migrate from one node to another on a demand basis, but pages are not replicated. Therefore, only one copy of a page containing a migratory variable exists in the system.

Migratory variables are handled efficiently by integrating their movement with that of the lock associated with them. That is, the lock and the variable are sent together in a single message to the location of the next process that is given the lock for accessing it.

3. *Write-shared.* A programmer may use this annotation with a shared variable to indicate to the system that the variable is updated concurrently by multiple processes, but the processes do not update the same parts of the variable. For example, in a matrix, different processes can concurrently update different row/column elements, with each process updating only the elements of one row/column. Munin avoids the false sharing problem of write-shared variables by allowing them to be concurrently updated by multiple processes.

A write-shared variable is replicated on all nodes where a process sharing is located. That is, when access to such a variable causes a network page fault to occur, the page having the variable is copied to the faulting node from one of its current nodes. If the access is a write access, the system first makes a copy of the page (called *twin page*) and then updates the original page. The process may perform several writes to the page before releasing it. When the page is released, the system performs a word-by-word comparison of the original page and the twin page and sends the differences to all nodes having a replica of the page.

When a node receives the differences of a modified page, the system checks if the local copy of the page was also modified. If not, the local copy of the page is updated by incorporating the received differences in it. On the other hand, if the local copy of the page was also modified, the local copy, its twin, and the received differences are compared word by word. If the same word has not been modified in both the local and remote copies of the page, the words of the original local page are updated by using the differences received from the remote node. On the other hand, if the comparison indicates that the same word was updated in both the local and remote copies of the page, a conflict occurs, resulting in a runtime error.

4. *Producer-consumer.* Shared variables that are written (produced) by only one process and read (consumed) by a fixed set of other processes may be annotated to be of producer-consumer type. Munin uses an "eager object movement" mechanism for this type of variable. In this mechanism, a variable is moved from the producer's node to the nodes of the consumers in advance of when they are required so that no network page fault occurs when a consumer accesses the variable. Moreover, the write-update protocol is used to update existing replicas of the variable whenever the producer updates the variable. If desired, the producer may send several updates together by using the locking mechanism. In this case, the procedure acquires a synchronization lock, makes several updates on the variable, and then releases the lock when the variable or the updates are sent to the nodes of consumer processes.

5. *Result.* Result variables are just the opposite of producer-consumer variables in the sense that they are written by multiple processes but read by only one process. Different processes write to different parts of the variable that do not conflict. The variable is read only when all its parts have been written. For example, in an application there may be different "worker" processes to generate and fill the elements of each row/column of a matrix, and once the matrix is complete, it may be used by a "master" process for further processing.

Munin uses a special write-update protocol for result variables in which updates are sent only to the node having the master process and not to all replica locations of the

variable (each worker process node has a replica of the variable). Since writes to the variable by different processes do not conflict, all worker processes are allowed to perform write operations concurrently. Moreover, since result variables are not read until all parts have been written, a worker process releases the variable only when it has finished writing all parts that it is supposed to write, when all updates are sent together in a single message to the master process node.

6. *Reduction.* Shared variables that must be atomically modified may be annotated to be of reduction type. For example, in a parallel computation application, a global minimum must be atomically fetched and modified if it is greater than the local minimum. In Munin, a reduction variable is always modified by being locked (acquire lock), read, updated, and unlocked (release lock). For better performance, a reduction variable is stored at a fixed owner that receives updates to the variable from other processes, synchronizes the updates received from different processes, performs the updates on the variable, and propagates the updated variable to its replica locations.

7. *Conventional.* Shared variables that are not annotated as one of the above types are conventional variables. The already described release consistency protocol of Munin is used to maintain the consistency of replicated conventional variables. The write invalidation protocol is used in this case to ensure that no process ever reads a stale version of a conventional variable. The page containing a conventional variable is dynamically moved to the location of a process that wants to perform a write operation on the variable.

Experience with Munin has shown that read-only, migratory, and write-shared annotation types are very useful because variables of these types are frequently used, but producer-consumer, result, and reduction annotation types are of little use because variables of these types are less frequently used.

5.7 REPLACEMENT STRATEGY

In DSM systems that allow shared-memory blocks to be dynamically migrated/replicated, the following issues must be addressed when the available space for caching shared data fills up at a node:

1. Which block should be replaced to make space for a newly required block?
2. Where should the replaced block be placed?

5.7.1 Which Block to Replace

The problem of replacement has been studied extensively for paged main memories and shared-memory multiprocessor systems. The usual classification of replacement algorithms group them into the following categories [Smith 1982]:

1. *Usage based versus non-usage based.* Usage-based algorithms keep track of the history of usage of a cache line (or page) and use this information to make replacement decisions. That is, the reuse of a cache line normally improves the replacement status of that line. Least recently used (LRU) is an example of this type of algorithm. Conversely, non-usage-based algorithms do not take the record of use of cache lines into account when doing replacement. First in, first out (FIFO) and Rand (random or pseudorandom) belong to this class.

2. *Fixed space versus variable space.* Fixed-space algorithms assume that the cache size is fixed while variable-space algorithms are based on the assumption that the cache size can be changed dynamically depending on the need. Therefore, replacement in fixed-space algorithms simply involves the selection of a specific cache line. On the other hand, in a variable-space algorithm, a fetch does not imply a replacement, and a swap-out can take place without a corresponding fetch.

Variable-space algorithms are not suitable for a DSM system because each node's memory that acts as cache for the virtually shared memory is fixed in size. Moreover, as compared to non-usage-based algorithms, usage-based algorithms are more suitable for DSM systems because they allow to take advantage of the data access locality feature. However, unlike most caching systems, which use a simple LRU policy for replacement, most DSM systems differentiate the status of data items and use a priority mechanism. As an example, the replacement policy used by the DSM system of IVY [Li 1986, 1988] is presented here. In the DSM system of IVY, each memory block of a node is classified into one of the following five types:

1. *Unused.* A free memory block that is not currently being used.
2. *Nil.* A block that has been invalidated.
3. *Read-only.* A block for which the node has only read access right.
4. *Read-owned.* A block for which the node has only read access right but is also the owner of the block.
5. *Writable.* A block for which the node has write access permission. Obviously, the node is the owner of the block because IVY uses the write-invalidate protocol.

Based on this classification of blocks, the following replacement priority is used:

1. Both unused and nil blocks have the highest replacement priority. That is, they will be replaced first if a block is needed. It is obvious for an unused block to have the highest replacement priority. A nil block also has the same replacement priority because it is no longer useful and future access to the block would cause a network fault to occur. Notice that a nil block may be a recently referenced block, and this is exactly why a simple LRU policy is not adequate.

2. The read-only blocks have the next replacement priority. This is because a copy of a read-only block is available with its owner, and therefore it is possible to simply discard that block. When the node again requires that block in the future, the block has to be brought from its owner node at that time.

3. Read-owned and writable blocks for which replica(s) exist on some other node(s) have the next replacement priority because it is sufficient to pass ownership to one of the replica nodes. The block itself need not be sent, resulting in a smaller message.

4. Read-owned and writable blocks for which only this node has a copy have the lowest replacement priority because replacement of such a block involves transfer of the block's ownership as well as the block from the current node to some other node. An LRU policy is used to select a block for replacement when all the blocks in the local cache have the same priority.

5.7.2 Where to Place a Replaced Block

Once a memory block has been selected for replacement, it should be ensured that if there is some useful information in the block, it should not be lost. For example, simply discarding a block having unused, nil, or read-only status does not lead to any loss of data. Similarly, discarding a read-owned or a writable block for which replica(s) exist on some other node(s) is also harmless. However, discarding a read-owned or a writable block for which there is no replica on any other node may lead to loss of useful data. Therefore, care must be taken to store them somewhere before discarding. The two commonly used approaches for storing a useful block at the time of its replacement are as follows:

1. *Using secondary store.* In this method, the block is simply transferred on to a local disk. The advantage of this method is that it does not waste any memory space, and if the node wants to access the same block again, it can get the block locally without a need for network access.

2. *Using the memory space of other nodes.* Sometimes it may be faster to transfer a block over the network than to transfer it to a local disk. Therefore, another method for storing a useful block is to keep track of free memory space at all nodes in the system and to simply transfer the replaced block to the memory of a node with available space. This method requires each node to maintain a table of free memory space in all other nodes. This table may be updated by having each node piggyback its memory status information during normal traffic.

5.8 THRASHING

Thrashing is said to occur when the system spends a large amount of time transferring shared data blocks from one node to another, compared to the time spent doing the useful work of executing application processes. It is a serious performance problem with DSM systems that allow data blocks to migrate from one node to another. Thrashing may occur in the following situations:

1. When interleaved data accesses made by processes on two or more nodes causes a data block to move back and forth from one node to another in quick succession (a ping-pong effect)

2. When blocks with read-only permissions are repeatedly invalidated soon after they are replicated

Such situations indicate poor (node) locality in references. If not properly handled, thrashing degrades system performance considerably. Therefore, steps must be taken to solve this problem. The following methods may be used to solve the thrashing problem in DSM systems:

1. *Providing application-controlled locks.* Locking data to prevent other nodes from accessing that data for a short period of time can reduce thrashing. An application-controlled lock can be associated with each data block to implement this method.

2. *Nailing a block to a node for a minimum amount of time.* Another method to reduce thrashing is to disallow a block to be taken away from a node until a minimum amount of time t elapses after its allocation to that node. The time t can either be fixed statically or be tuned dynamically on the basis of access patterns. For example, Mirage [Fleisch and Popek 1989] employs this method to reduce thrashing and dynamically determines the minimum amount of time for which a block will be available at a node on the basis of access patterns.

The main drawback of this scheme is that it is very difficult to choose the appropriate value for the time t. If the value is fixed statically, it is liable to be inappropriate in many cases. For example, if a process accesses a block for writing to it only once, other processes will be prevented from accessing the block until time t elapses. On the other hand, if a process accesses a block for performing several write operations on it, time t may elapse before the process has finished using the block and the system may grant permission to another process for accessing the block. Therefore, tuning the value of t dynamically is the preferred approach. In this case, the value of t for a block can be decided based on past access patterns of the block. The MMU's reference bits may be used for this purpose. Another factor that may be used for deciding the value of t for a block is the length of the queue of processes waiting for their turn to access the block.

3. *Tailoring the coherence algorithm to the shared-data usage patterns.* Thrashing can also be minimized by using different coherence protocols for shared data having different characteristics. For example, the coherence protocol used in Munin for write-shared variables avoids the false sharing problem, which ultimately results in the avoidance of thrashing.

Notice from the description above that complete transparency of distributed shared memory is compromised somewhat while trying to minimize thrashing. This is because most of the approaches described above require the programmer's assistance. For example, in the method of application-controlled locks, the use of locks needs to be directed toward a particular shared-memory algorithm and hence the shared-memory abstraction can no longer be transparent. Moreover, the application must be aware of the shared data it is accessing and its shared access patterns. Similarly, Munin requires programmers to annotate shared variables with standard annotation types, which makes the shared-memory abstraction nontransparent.

5.9 OTHER APPROACHES TO DSM

Depending on the manner in which data caching (placement and migration of data) is managed, there are three main approaches for designing a DSM system:

1. Data caching managed by the operating system
2. Data caching managed by the MMU hardware
3. Data caching managed by the language runtime system

This being a book on operating systems, in this chapter we mainly concentrate on the first approach. Systems such as IVY [Li 1986] and Mirage [Fleish and Popek 1989] fall in this category. In these systems, each node has its own memory and access to a word in another node's memory causes a trap to the operating system. The operating system then fetches and acquires the page containing the accessed word from the remote node's memory by exchanging messages with that machine. Therefore, in these systems, the placement and migration of data are handled by the operating system. For completion, the other two approaches of designing a DSM system are briefly described below.

The second approach is to manage caching by the MMU. This approach is used in multiprocessors having hardware caches. In these systems, the DSM implementation is done either entirely or mostly in hardware. For example, if the multiprocessors are interconnected by a single bus, their caches are kept consistent by snooping on the bus. In this case, the DSM is implemented entirely in hardware. The DEC Firefly workstation belongs to this category. On the other hand, if the multiprocessors are interconnected by switches, directories are normally used in addition to hardware caching to keep track of which CPUs have which cache blocks. Algorithms used to keep cached data consistent are stored mainly in MMU microcode. Therefore, in this case, the DSM is implemented mostly in hardware. Stanford's Dash [Lenoski et al. 1992] and MIT's Alewife [Agarwal et al. 1991, Kranz et al. 1993] systems belong to this category. Notice that in these systems, when a remote access is detected, a message is sent to the remote memory by the cache controller or MMU (not by the operating system software).

The third approach is to manage caching by the language runtime system. In these systems, the DSM is structured not as a raw linear memory of bytes from 0 to total size of the combined memory of all machines, but as a collection of programming language constructs, which may be shared variables and data structures (in conventional programming languages) or shared objects (in object-oriented programming languages). In these systems, the placement and migration of shared variables/objects are handled by the language runtime system in cooperation with the operating system. That is, when a variable/object is accessed by a process, it is the responsibility of the runtime system and the operating system to successfully perform the requested access operation on the variable/object independently of its current location. An advantage of this approach is that programming languages may be provided with features to allow programmers to specify the usage pattern of shared variables/objects

for their applications, and the system can support several consistency protocols and use the one most suitable for a shared variable/object. Therefore, these systems can allow consistency protocols to be applied at the granularity of individual data items and can rely on weaker consistency models than sequential consistency model for better concurrency. However, a drawback of this approach is that it imposes extra burden on programmers. Munin [Bennett et al. 1990] and Midway [Bershad et al. 1993] are examples of systems that structure their DSM as a collection of shared variables and data structures. On the other hand, Orca [Bal et al. 1992] and Linda [Carriero and Gelernter 1989] are examples of systems that structure their DSM as a collection of shared objects.

5.10 HETEROGENEOUS DSM

Computers of different architectures normally have different characteristic features. For example, supercomputers and multiprocessors are good at compute-intensive applications while personal computers and workstations usually have good user interfaces. A heterogeneous computing environment allows the applications to exploit the best of all the characteristic features of several different types of computers. Therefore, heterogeneity is often desirable in distributed systems.

Heterogeneous DSM is a mechanism that provides the shared-memory paradigm in a heterogeneous environment and allows memory sharing among machines of different architectures. At first glance, sharing memory among machines of different architectures seems almost impossible. However, based on the measurements made on their experimental prototype heterogeneous DSM, called Mermaid, Zhou et al. [1990, 1992] have concluded that heterogeneous DSM is not only feasible but can also be comparable in performance to its homogeneous counterpart (at least for some applications). The two main issues in building a DSM system on a network of heterogeneous machines are data conversion and selection of block size. These issues are described below. The following description is based on the material presented in [Zhou et al. 1990, 1992].

5.10.1 Data Conversion

Machines of different architectures may use different byte orderings and floating-point representations. When data is transferred from a node of one type to a node of another type, it must be converted before it is accessed on the destination node. However, the unit of data transfer in a DSM system is normally a block. Therefore, when a block is migrated between two nodes of different types, the contents of the block must be converted. But the conversion itself has to be based on the types of data stored in the block. It is not possible for the DSM system to convert a block without knowing the type of application-level data contained in the block and the actual block layout. Therefore, it becomes necessary to take assistance from application programmers, who know the layout of the memory being used by their application programs. Two approaches proposed in the literature for data conversion in a heterogeneous DSM are described below.

Structuring the DSM System as a Collection of Source Language Objects

In this method, the DSM system is structured as a collection of variables or objects in the source language so that the unit of data migration is an object instead of a block. A DSM compiler is used that adds conversion routines for translating variables/objects in the source language among various machine architectures. For each access to a shared-memory object from a remote node, a check is made by the DSM system if the machine of the node that holds the object and the requesting node are compatible. If not, a suitable conversion routine is used to translate the object before migrating it to the requesting node.

This method of data conversion is used in the Agora shared-memory system [Bisiani et al. 1987]. In Agora, each machine marks messages with a machine tag. A message contains an object being migrated from one node to another. Messages between identical or compatible machines do not require translation. When translation is necessary, the type of information associated with the destination context is used to do a one-pass translation of the object. Depending on the field tag and the machine requirements for data alignment, the translation process may involve such operations as field-by-field swapping of bytes, inserting/removing gaps where necessary, and shuffling of bits.

Structuring a DSM system as a collection of source language objects may be useful from the data conversion point of view but is generally not recommended due to performance reasons. This is because objects in conventional programming languages typically are scalars (single-memory words), arrays, and structures. None of these types is suitable as a shared entity of a DSM system. For each shared entity, access rights must be issued for every entity and the migration of an entity involves communication overhead. Therefore, the choice of scalar data types as a unit of sharing would make a system very inefficient. On the other hand, arrays may easily be too large to be treated as units of sharing and data migration. Large data objects lead to false sharing and thrashing, and their migration often requires fragmentation and reassembly operations to be carried out on them due to the limited packet size of transport protocols.

Allowing Only One Type of Data in a Block

This mechanism is used in Mermaid [Zhou et al. 1990, 1992], which uses a page size as its block size. Therefore, a page can contain only one type of data. In Mermaid, the DSM page (block) table entry keeps additional information identifying the type of data maintained in that page and the amount of data allocated to the page. Whenever a page is moved between two machines of different architectures, a routine converts the data in the page to the appropriate format. Mermaid requires the application programmers to provide the conversion routine for each user-defined data type in the application. Mermaid designers pointed out that the cost of converting a page is small compared to the overall cost of page migration.

Zhou et al. [1990] identified the following limitations of this approach:

1. Allowing a page to contain data of only one type may lead to wastage of memory due to fragmentation, resulting in increased paging activity.

2. The mechanism requires that the compilers used on different types of machines must be compatible in the sense that in the compiler-generated code the size of each data type and the order of the fields within compound structures must be the same on each machine. If incompatible compilers are used for an application to generate code for two different machines such that the size of the application-level data structures differs for the two machines, the mapping between pages on the two machines would not be one to one. That is, it may not be possible for a structure to fit on a page in its entirety after the conversion or, conversely, some data from the following page may be needed in order to complete the conversion of the current page [Zhou et al. 1990]. This complicates the conversion process.

3. Another problem associated with this method is that entire pages are converted even though only a small portion may be accessed before it is transferred away. Sometimes this problem may be more severe in the first method, which converts an entire object. For example, a large array may occupy several pages of memory, and migration of this object would convert the data in all the occupied pages even when only the first few array elements may be accessed before the object is transferred away.

4. The mechanism is not fully transparent because it requires the users to provide the conversion routines for user-defined data types and a table specifying the mapping of data types to conversion routines. The transparency problem may be solved by automatically generating the conversion routines and the mapping table by using a preprocessor on the user program [Zhou et al. 1990].

Another serious problem associated with the data conversion issue in heterogeneous DSM systems is that of the accuracy of floating-point values in numerical applications. Since an application has no control over how often a data is migrated or converted, numerical accuracy of floating-point data may be lost if the data is converted several times and the results may become numerically questionable.

5.10.2 Block Size Selection

Recall that in a homogeneous DSM system the block size is usually the same size as a native virtual memory (VM) page, so that the MMU hardware can be used to trigger a DSM block fault. However, in a heterogeneous environment, the virtual memory page size may be different for machines of different types. Therefore, block size selection becomes a complicated task in such a situation. Zhou et al. [1990, 1992] proposed the use of one of the following algorithms for block size selection in a heterogeneous DSM system:

1. *Largest page size algorithm.* In this method, the DSM block size is taken as the largest VM page size of all machines. Since VM page sizes are normally powers of 2, multiple smaller VM pages fit exactly in one DSM block. If a page fault occurs on a node with a smaller page size, it will receive a block of multiple pages that includes the desired page. This algorithm suffers from the same false sharing and thrashing problems associated with large-sized blocks.

2. *Smallest page size algorithm.* In this method, the DSM block size is taken as the smallest VM page size of all machines. If a page fault occurs on a node with a larger page size, multiple blocks (whose total size is equal to the page size of the faulting node) are moved to satisfy the page fault. Although this algorithm reduces data contention, it suffers from the increased communication and block table management overheads associated with small-sized blocks.

3. *Intermediate page size algorithm.* To balance between the problems of large- and small-sized blocks, a heterogeneous DSM system may select to choose a block size somewhere in between the largest VM page size and the smallest VM page size of all machines.

5.11 ADVANTAGES OF DSM

Distributed Shared Memory is a high-level mechanism for interprocess communication in loosely coupled distributed systems. It is receiving increased attention because of the advantages it has over the message-passing mechanisms. These advantages are discussed below.

5.11.1 Simpler Abstraction

By now it is widely recognized that directly programming loosely coupled distributed-memory machines using message-passing models is tedious and error prone. The main reason is that the message-passing models force programmers to be conscious of data movement between processes at all times, since processes must explicitly use communication primitives and channels or ports. To alleviate this burden, RPC was introduced to provide a procedure call interface. However, even in RPC, since the procedure call is performed in an address space different from that of the caller's address space, it is difficult for the caller to pass context-related data or complex data structures; that is, parameters must be passed by value. In the message-passing model, the programming task is further complicated by the fact that data structures passed between processes in the form of messages must be packed and unpacked. The shared-memory programming paradigm shields the application programmers from many such low-level concerns. Therefore, the primary advantage of DSM is the simpler abstraction it provides to the application programmers of loosely coupled distributed-memory machines.

5.11.2 Better Portability of Distributed Application Programs

The access protocol used in case of DSM is consistent with the way sequential applications access data. This allows for a more natural transition from sequential to distributed applications. In principle, distributed application programs written for a shared-memory multiprocessor system can be executed on a distributed shared-memory system without change. Therefore, it is easier to port an existing distributed application

program to a distributed-memory system with DSM facility than to a distributed-memory system without this facility.

5.11.3 Better Performance of Some Applications

The layer of software that provides DSM abstraction is implemented on top of a message-passing system and uses the services of the underlying message-passing communication system. Therefore, in principle, the performance of applications that use DSM is expected to be worse than if they use message-passing directly. However, this is not always true, and it has been found that some applications using DSM can even outperform their message-passing counterparts. This is possible for three reasons [Stumm and Zhou 1990]:

1. *Locality of data.* The computation model of DSM is to make the data more accessible by moving it around. DSM algorithms normally move data between nodes in large blocks. Therefore, in those applications that exhibit a reasonable degree of locality in their data accesses, communication overhead is amortized over multiple memory accesses. This ultimately results in reduced overall communication cost for such applications.

2. *On-demand data movement.* The computation model of DSM also facilitates on-demand movement of data as they are being accessed. On the other hand, there are several distributed applications that execute in phases, where each computation phase is preceded by a data-exchange phase. The time needed for the data-exchange phase is often dictated by the throughput of existing communication bottlenecks. Therefore, in such applications, the on-demand data movement facility provided by DSM eliminates the data-exchange phase, spreads the communication load over a longer period of time, and allows for a greater degree of concurrency.

3. *Larger memory space.* With DSM facility, the total memory size is the sum of the memory sizes of all the nodes in the system. Thus, paging and swapping activities, which involve disk access, are greatly reduced.

5.11.4 Flexible Communication Environment

The message-passing paradigm requires recipient identification and coexistence of the sender and receiver processes. That is, the sender process of a piece of data must know the names of its receiver processes (except in multicast communication), and the receivers of the data must exist at the time the data is sent and in a state that they can (or eventually can) receive the data. Otherwise, the data is undeliverable. In contrast, the shared-memory paradigm of DSM provides a more flexible communication environment in which the sender process need not specify the identity of the receiver processes of the data. It simply places the data in the shared memory and the receivers access it directly from the shared memory. Therefore, the coexistence of the sender and receiver processes is also not necessary in the shared-memory paradigm. In fact, the lifetime of the shared data is independent of the lifetime of any of its receiver processes.

5.11.5 Ease of Process Migration

Migration of a process from one node to another in a distributed system (described in Chapter 8) has been shown to be tedious and time consuming due to the requirement of transferring the migrant process's address space from its old node to its new node. However, the computation model of DSM provides the facility of on-demand migration of data between processors. This facility allows the migrant process to leave its address space on its old node at the time of migration and fetch the required pages from its new node on demand at the time of accessing. Hence in a distributed system with DSM facility, process migration is as simple as detaching the process control block (PCB) of the migrant process from the processor of the old node and attaching it to the ready queue of the new node's processor. A PCB is a data block or a record associated with each process that contains useful information such as process state, CPU registers, scheduling information, memory management information, I/O status information, and so on. This approach provides a very natural and efficient form of process migration between processors in a distributed system.

5.12 SUMMARY

Programming of applications for loosely coupled distributed-memory machines with the message-passing paradigm is a difficult and error-prone task. The Distributed Shared Memory (DSM) facility simplifies this programming task by providing a higher level abstraction that allows programmers to write programs with the shared-memory paradigm, which is consistent with the way sequential applications access data.

Important issues involved in the design and implementation of a DSM system are granularity of data sharing, structure of the shared-memory space, memory coherence and access synchronization, data location and access, replacement strategy, handling of thrashing, and heterogeneity.

Granularity refers to block size, which is the unit of data sharing and data transfer across the network. Both large- and small-sized blocks have their own advantages and limitations. Several DSM systems choose the virtual memory page size as block size so that the MMU hardware can be used to trigger a DSM block fault.

The structure of the shared-memory space of a DSM system defines the abstract view to be presented to application programmers of that system. The three commonly used methods for structuring the shared-memory space of a DSM system are no structuring, structuring by data type, and structuring as a database. The structure and granularity of a DSM system are closely related.

Memory coherence is an important issue in the design of a DSM system. It deals with the consistency of a data block lying in the main memories of two or more nodes of the system. Several consistency models have been proposed in the literature to handle this issue. The main ones described in this chapter are strict consistency, sequential consistency, causal consistency, PRAM consistency, processor consistency, weak consistency, and release consistency. Of these, sequential consistency and release consistency are appropriate for a large number of applications.

Protocols for implementing the sequential consistency model in a DSM system depend on the following four replication and migration strategies used by the DSM system:

1. Nonreplicated, nonmigrating blocks (NRNMBs)
2. Nonreplicated, migrating blocks (NRMBs)
3. Replicated, migrating blocks (RMBs)
4. Replicated, nonmigrating blocks (RNMBs)

Protocols for implementing sequential consistency for each of these strategies have been described in this chapter. The data-locating mechanisms suitable for each case have also been described. In addition, the Munin system has been presented as an example of a release consistent DSM system.

Replacement strategy deals with the selection of a block to be replaced when the available space for caching shared data fills up and the placement of a replaced block. The usual classifications of replacement algorithms are usage-based versus non-usage-based and fixed-space versus variable-space. DSM systems use fixed-space, usage-based replacement algorithms. At the time of its replacement, a useful block may be placed in either the secondary storage of the local node or the memory space of some other node.

Thrashing is a serious performance problem with DSM systems that allow data blocks to migrate from one node to another. Methods that may be used to solve the thrashing problem in DSM systems are providing application-controlled locks, nailing a block to a node for a minimum amount of time, and tailoring the coherence algorithm to the shared-data usage patterns.

Depending on the manner in which data caching is managed, there are three main approaches to designing a DSM system: data caching managed by the operating system, data caching managed by the MMU hardware, and data caching managed by the language runtime system.

Heterogeneous DSM is a mechanism that provides the shared-memory paradigm in a heterogeneous environment and allows memory sharing among machines with different architectures. The two main issues in building this facility are data conversion and selection of block size. Two approaches proposed in the literature for data conversion in a heterogenous DSM are structuring the DSM system as a collection of source language objects and allowing only one type of data in a DSM block. Three algorithms that may be used for block size selection in a heterogeneous DSM system are largest page size algorithm, smallest page size algorithm, and intermediate page size algorithm.

Research has shown that DSM systems are viable, and they have several advantages over the message-passing systems. However, they are still far from mature. Most existing DSM systems are very small experimental or prototype systems consisting of only a few nodes. The performance results to date are also preliminary and based on a small group of applications or a synthetic workload. Nevertheless, research has proved that DSM effectively supports parallel processing and promises to be a fruitful and exciting area of research for the coming decade.

EXERCISES

5.1. The distributed shared-memory abstraction is implemented by using the services of the underlying message-passing communication system. Therefore, in principle, the performance of applications that use DSM is expected to be worse than if they use message passing directly. In spite of this fact, why do some distributed operating system designers support the DSM abstraction in their systems? Are there any applications that can have better performance in a system with DSM facility than in a system that has only message-passing facility? If yes, give the types of such applications. If no, explain why.

5.2. Discuss the relative advantages and disadvantages of using large block size and small block size in the design of a block-based DSM system. Why do most DSM system designers prefer to use the typical page size used in a conventional virtual-memory implementation as the block size of the DSM system?

5.3. It is often said that the structure of the shared-memory space and the granularity of data sharing in a DSM system are closely related. Explain why.

5.4. What is false sharing? When is it likely to occur? Can this problem lead to any other problem in a DSM system? Give reasons for your answer.

5.5. What should be done to minimize the false sharing problem? Can this problem be completely eliminated? What other problems may occur if one tries to completely eliminate the false sharing problem?

5.6. Discuss the relative advantages and disadvantages of using the NRNMB, NRMB, RMB, and RNMB strategies in the design of a DSM system.

5.7. Discuss the relative advantages and disadvantages of the various data-locating mechanisms that may be used in a DSM system that uses the NRMB strategy.

5.8. A sequentially consistent DSM system uses the RMB strategy. It employs the write-invalidate approach for updating data blocks and the centralized-server algorithm for locating the owner of a block and keeping track of the nodes that currently have a valid copy of the block. Write pseudocode for the implementation of the memory coherence scheme of this DSM system.

5.9. Why is a global sequencer needed in a sequentially consistent DSM system that employs the write-update protocol?

5.10. Most DSM systems in which caching is managed by the operating system use the write-invalidate scheme for consistency instead of the write-update scheme. Explain why.

5.11. Differentiate between weak consistency and release consistency. Which of the two will you prefer to use in the design of a DSM system? Give reasons for your answer.

5.12. A programmer is writing an application for a release-consistent DSM system. However the application needs sequential consistency to produce correct results. What precautions must the programmer take?

5.13. Differentiate between PRAM consistency and processor consistency.

5.14. Give the relative advantages and disadvantages of sequential and release consistency models.

5.15. What is causal consistency? Give an example of an application for which causal consistency is the most suitable consistency model.

5.16. Propose a suitable replacement algorithm for a DSM system whose shared-memory space is structured as objects. One of the goals in this case may be to minimize memory fragmentation.

5.17. Why does the simple LRU policy often used for replacing cache lines in a buffer cache not work well as a replacement policy for replacing blocks in a DSM system?

5.18. To handle the issue of where to place a replaced block, the DSM system of Memnet [Delp 1988] uses the concept of "home memory," in which each block has a home memory. When replacement of a block requires that the block be transferred to some other node's memory, the block is transferred to the node whose memory is the home memory for the block. What are the advantages and disadvantages of this approach as compared to the one presented in this chapter?

5.19. What are the main causes of thrashing in a DSM system? What are the commonly used methods to solve the trashing problem in a DSM system?

5.20. Complete transparency of a DSM system is compromised somewhat when a method is used to minimize thrashing. Therefore, the designer of a DSM system is of the opinion that instead of using a method to minimize thrashing, a method should be used by which the system automatically detects and resolves this problem. Propose a method by which the system can detect thrashing and a method to resolve it once it has been detected.

5.21. A distributed system has DSM facility. The process-scheduling mechanism of this system selects another process to run when a fault occurs for the currently running process, and the CPU is utilized while the block is being fetched. Two system engineers arguing about how to better utilize the CPUs of this system have the following opinions:

(a) The first one says that if a large number of processes are scheduled for execution at a node, the available memory space of the node can be distributed among these processes so that almost always there will be a ready process to run when a page fault occurs. Thus, CPU utilization can be kept high.

(b) The second one says that if only a few processes are scheduled for execution at a node, the available memory space of the node can be allocated to each of the few processes, and each process will produce fewer page faults. Thus, CPU utilization can be kept high.

Whose argument is correct? Give reasons for your answer.

5.22. What are the three main approaches for designing a DSM system?

5.23. What are some of the issues involved in building a DSM system on a network of heterogeneous machines? Suggest suitable methods for handling these issues.

5.24. Are DSM systems suitable for both LAN and WAN environments? Give reasons for your answer.

5.25. Suggest some programming practices that will reduce network block faults in a DSM system.

5.26. Write pseudocode descriptions for handling a block fault in each of the following types of DSM systems:

(a) A DSM system that uses the NRNMB strategy
(b) A DSM system that uses the NRMB strategy
(c) A DSM system that uses the RMB strategy
(d) A DSM system that uses the RNMB strategy

You can make any assumptions that you feel necessary, but state the assumptions made.

BIBLIOGRAPHY

[Adve and Hill 1990] Adve, S., and Hill, M., "Weak Ordering: A New Definition," In: *Proceedings of the 17th International Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, pp. 2–14 (1990).

[Agarwal et al. 1991] Agarwal, A., Chaiken, D., D'Souza, G., Johnson, K., Kranz, D., Kubiatowicz, J., Kurihara, K., Lim, B., Maa, G., Nussbaum, D., Parkin, M., and Yeung, D., "The MIT Alewife Machine: A Large-Scale Distributed Memory Multiprocessor," In: *Proceedings of the Workshop on Scalable Shared Memory Multiprocessors*, Kluwer Academic, Norwell, MA (1991).

[Bal 1991] Bal, H. E., *Programming Distributed Systems*, Prentice-Hall, London, England (1991).

[Bal et al. 1992] Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. SE-18, pp. 190–205 (1992).

[Baldoni et al. 1995] Baldoni, R., Mostefaoui, A., and Raynal, M., "Efficient Causally Ordered Communications for Multimedia Real Time Applications," In: *Proceedings of the 4th International Symposium on High Performance Distributed Computing*, IEEE, New York (1995).

[Bennett et al. 1990] Bennett, J., Carter, J., and Zwaenepoel, W., "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," In: *Proceedings of the 1990 Conference on Principles and Practice of Parallel Programming*, Association for Computing Machinery, New York, pp. 168–176 (1990).

[Bershad et al. 1993] Bershad, B. N., Zekauskas, M. J., and Sawdon, W. A., "The Midway Distributed Shared Memory System," In: *Proceedings of the IEEE COMPCON Conference*, IEEE, New York, pp. 528–537 (1993).

[Bisiani and Ravishankar 1990] Bisiani, R., and Ravishankar, M., "Plus: A Distributed Shared-Memory System," In: *Proceedings of the 17th International Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, pp. 115–124 (1990).

[Bisiani et al. 1987] Bisiani, R., Alleva, F., Correrini, F., Forin, A., Lecouat, F., and Lerner, R., "Heterogeneous Parallel Processing: The Agora Shared Memory," Technical Report No. CMU-CS-87–112, Computer Science Department, Carnegie-Mellon University (March 1987).

[Bisiani et al. 1989] Bisiani, R., Nowatzyk, A., and Ravishankar, M., "Coherent Shared Memory on a Distributed Memory Machine," In: *Proceedings of the International Conference on Parallel Processing*, IEEE, New York, pp. 133–141 (August 1989).

[Campine et al. 1990] Campine, G. A., Geer, Jr., D. E, and Ruh, W. N., "Project Athena as a Distributed Computer System," *IEEE Computer*, Vol. 23, pp. 40–51 (1990).

[Carriero and Gelernter 1989] Carriero, N., and Gelernter, D., "Linda in Context," *Communications of the ACM*, Vol. 32, No. 4, pp. 444–458 (1989).

[Carriero et al. 1986] Carriero, N., Gelernter, D., and Leichter, J., "Distributed Data Structures in Linda," In: *Proceedings of the ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery, New York (1986).

[Carter et al. 1991] Carter, J. B., Bennett, J. K., and Zwaenepoel, W., "Implementation and Performance of Munin," In: *Proceedings of the 13th Symposium on Operating Systems Principles*, Association for Computing Machinery, New York, NY, pp. 152–164 (1991).

[Carter et al. 1994] Carter, J. B., Bennett, J. K., and Zwaenepoel, W., "Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems," *ACM Transactions on Computer Systems*, Vol. 12 (1994).

[Chase et al. 1989] Chase, J. S., Amador, F. G., Lazowska, E. D., Levy, H. M., and Littlefield, R. J., "The Amber System: Parallel Programming on a Network of Multiprocessors," In: *Proceedings of the 12th Symposium on Operating Systems Principles*, Association for Computing Machinery, New York, NY, pp. 147–158 (1989).

[Cheong and Veidenbaum 1988] Cheong, H., and Veidenbaum, A. V., "A Cache Coherence Scheme with Fast Selective Invalidation," In: *Proceedings of the 15th International Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, pp. 299–307 (1988).

[Cheriton 1986] Cheriton, D. R., "Problem-Oriented Shared Memory: A Decentralized Approach to Distributed System Design," In: *Proceedings of the 6th International Conference on Distributed Computing Systems*, IEEE, New York, pp. 190–197 (May 1986).

[Cheriton et al. 1991] Cheriton, D. R., Goosen, H. A., and Boyle, P. D., "Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture," *IEEE Computer*, Vol. 24, No. 2, pp. 33–46 (1991).

[Cox and Fowler 1989] Cox, A. L., and Fowler, R. J., "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM," In: *Proceedings of the 12th Symposium on Operating Systems Principles*, Association for Computing Machinery, New York, NY, pp. 32–34 (December 1989).

[Dasgupta et al. 1991] Dasgupta, P., LeBlanc, R. J., Ahmad, Jr., M., and Ramachandran, U., "The Clouds Distributed Operating System," *IEEE Computer*, Vol. 24, No. 11, pp. 34–44 (1991).

[Delp 1988] Delp, G. S., "The Architecture and Implementation of MemNet: An Experiment on High-Speed Memory Mapped Network Interface," Ph.D. Dissertation, Department of Computer Science, University of Delaware (1988).

[Delp et al. 1991] Delp, G. S., Farber, D. J., Minnich, R. G., Smith, J. M., and Tam, M. C., "Memory as a Network Abstraction," *IEEE Network*, Vol. 5, pp. 34–41 (1991).

[Dubois et al. 1986] Dubois, M., Scheurich, C., and Briggs, F. A., "Memory Access Buffering in Multiprocessors," In: *Proceedings of the 13th Annual Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, pp. 343–442 (1986).

[Dubois et al. 1988] Dubois, M., Scheurich, C., and Briggs, F. A.,"Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, Vol. 21, No. 2, pp. 9–21 (1988).

[Fekete et al. 1995] Fekete, A, Kaashoek, F., and Lynch, N., "Providing Sequentially-Consistent Shared Objects Using Group and Point-to-Point Communication," In: *Proceedings of the 15th International Conference on Distributed Computing Systems*, IEEE, New York (May-June 1995).

[Fleisch 1987] Fleisch, B. D., "Distributed Shared Memory in a Loosely Coupled Distributed System," In: *Proceedings of the 1987 ACM SIGCOMM Workshop*, Association for Computing Machinery, New York, NY (1987).

[Fleisch and Popek 1989] Fleisch, B. D., and Popek, G. J., "Mirage: A Coherent Distributed Shared Memory Design," In: *Proceedings of the 12th ACM Symposium on Operating System Principles*, Association for Computing Machinery, New York, NY, pp. 211–223 (December 1989).

[Forin et al. 1989] Forin, A., Barrera, J., Young, M., and Rashid, R.,"Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach," In: *Proceedings of the 1989 Winter Usenix Conference* (January 1989).

[Frank 1984] Frank, S. J., "Tightly Coupled Multiprocessor System Speeds Memory-Access Times," *Electronics*, pp. 164–169 (January 1984).

[Garcia Molina and Wiederhold 1982] Garcia-Molina, H., and Wiederhold, G., "Read-Only Transactions in a Distributed Database," *ACM Transactions on Database Systems*, Vol. 7, No. 2, Association for Computing Machinery, New York, NY, pp. 209–234 (1982).

[Gharachorloo et al. 1990] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," In: *Proceedings of the 17th Annual Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, pp. 15–26 (1990).

[Gharachorloo et al. 1991] Gharachorloo, K., Gupta, A., and Hennessy, J., "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors," In: *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, IEEE Computer Society Press, Los Alamitos, CA, pp. 245–257 (1991).

[Ghose 1995] Ghose, K., "SNOW: Hardware Supported Distributed Shared Memory over a Network of Workstations," In: *Proceedings of the 24th Annual International Conference on Parallel Processing*, IEEE, New York (August 1995).

[Goodman 1983] Goodman, J. R., "Using Cache Memory to Reduce Processor-Memory Traffic," In: *Proceedings of the 10th Annual Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, pp. 124–131 (June 1983).

[Goodman 1989] Goodman, J. R., "Cache Consistency and Sequential Consistency," Technical Report No. 61, IEEE Scalable Coherent Interface Working Group, IEEE, New York (1989).

[Goodman et al. 1989] Goodman, J. R., Vernon, M. K., and Woest, P. J., "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," In: *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, IEEE Computer Society Press, Los Alamitos, CA, pp. 64–73 (1989).

[Harty and Cheriton 1992] Harty, K., and Cheriton, D., "Application-Controlled Physical Memory Using External Page-Cache Management," In: *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery, New York, NY, pp. 187–199 (1992).

[Hutto and Ahamad 1990] Hutto, P. W., and Ahamad, M., "Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories," In: *Proceedings of the 10th International Conference on Distributed Computing Systems*, IEEE, New York, pp. 302–311 (1990).

[Johnson et al. 1995] Johnson, D., Lilja, D., and Riedl, J., "A Circulating Active Barrier Synchronization Mechanism," In: *Proceedings of the 24th Annual International Conference on Parallel Processing*, IEEE, New York (August 1995).

[Katz et al. 1985] Katz, R. H., Eggers, S. J., Wood, D. A., Perkins, C. L., and Sheldon, R. G., "Implementing a Cache Consistency Protocol," In: *Proceedings of the 12th Annual Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, pp. 276–283 (June 1985).

[Keleher et al. 1992] Keleher, P., Cox, A. L., and Zwaenepoel, W., "Lazy Release Consistency," In: *Proceedings of the 19th International Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, pp. 13–21 (1992).

[Kessler and Livny 1989] Kessler, R. E., and Livny, M., "An Analysis of Distributed Shared Memory Algorithms," In: *Proceedings of the 9th International Conference on Distributed Computing Systems*, IEEE, New York, pp. 98–104 (June 1989).

[Kranz et al. 1993] Kranz, D., Johnson, K., Agarwal, A., Kubiatowicz, J. J., and Lim, B., "Integrating Message Passing and Shared Memory: Early Experiences," In: *Proceedings of the 4th Symposium on Principles and Practice of Parallel Programming*, Association for Computing Machinery, New York, NY, pp. 54–63 (May 1993).

[Lamport 1979] Lamport, L., "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, Vol. C-28, IEEE, New York, pp. 690–691 (1979).

[Lenoski and Weber 1995] Lenoski, D. E., and Weber, W. D., *Scalable Shared-Memory Multiprocessing*, Morgan Kaufmann, San Francisco, CA (1995).

[Lenoski et al. 1992] Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W. D., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. S., "The Stanford Dash Multiprocessor," *IEEE Computer*, Vol. 25, No. 3, pp. 63–79 (1992).

[Lenoski et al. 1993] Lenoski, D., Laudon, J., Joe, T., Nakahira, D., Steves, L., Gupta, A., and Hennessy, J., "The DASH Prototype: Logic Overhead and Performance," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 1, pp. 41–61 (1993).

[Li 1986] Li, K., "Shared Virtual Memory on Loosely Coupled Multiprocessors," Ph.D. Dissertation, Technical Report No. YALE/DCS/RR-492, Department of Computer Science, Yale University (September 1986).

[Li 1988] Li, K., "IVY: A Shared Virtual Memory System for Parallel Computing," In: *Proceedings of the International Conference on Parallel Processing*, IEEE, New York, pp. 94–101 (August 1988).

[Li and Hudak 1989] Li, K., and Hudak, P., "Memory Coherence in Shared Virtual Memory Systems," *ACM Transactions on Computing Systems*, Vol. 7, No. 4, pp. 321–359 (1989).

[Li and Schaefer 1989] Li, K., and Schaefer, R.,"A Hypercube Shared Virtual Memory System," In: *Proceedings of the International Conference on Parallel Processing*, Pennsylvania State University Press, pp. 125–132 (1989).

[Lilja 1993] Lilja, D. J., "Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons," *ACM Computing Surveys*, Vol. 25, pp. 303–338 (1993).

[Lipton and Sandberg 1988] Lipton, R. J., and Sandberg, J. S., "Pram: A Scalable Shared Memory," Technical Report No. CS-TR-180–88, Princeton University (1988).

[Liskov 1988] Liskov, B., "Distributed Programming in Argus," *Communications of the ACM*, Vol. 31, No. 3, pp. 300–313 (1988).

[Minnich and Farber 1989] Minnich, R. G., and Farber, D. J., "The Mether System: A Distributed Shared Memory for SunOS 4.0," In: *Proceedings of the 1989 Summer Usenix Conference*, pp. 51–60 (1989).

[Minnich and Farber 1990] Minnich, R. G., and Farber, D. J., "Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory," In: *Proceedings of the 10th International Conference on Distributed Computing Systems*, IEEE, New York (June 1990).

[Mosberger 1993] Mosberger, D., "Memory Consistency Models," Technical Report No. TR 93/11, Department of Computer Science, University of Arizona (1993).

[Nitzberg and Virginia Lo 1991] Nitzberg, N., and Virginia Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer*, Vol. 24, No. 11, pp. 52–60 (1991).

[Oguchi et al. 1995] Oguchi, M., Aida, H., and Saito, T., "A Proposal for a DSM Architecture Suitable for a Widely Distributed Environment and Its Evaluation," In: *Proceedings of the 4th International Symposium on High Performance Distributed Computing*, IEEE, New York (August 1995).

[Ramachandran and Khalidi 1989] Ramachandran, U., and Khalidi, M.Y. A., "An Implementation of Distributed Shared Memory," *First Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Usenix Association, Berkeley, CA, pp. 21–38 (1989).

[Sane et al. 1990] Sane, A., MacGregor, K., and Campbell, R., "Distributed Virtual Memory Consistency Protocols: Design and Performance," *Second IEEE Workshop on Experimental Distributed Systems*, IEEE, New York, pp. 91–96 (October 1990).

[Scheurich and Dubois 1988] Scheurich, C., and Dubois, M., "Dynamic Page Migration in Multiprocessors with Distributed Global Memory," In: *Proceedings of the 8th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, Los Alamitos, CA, pp. 162–169 (June 1988).

[Shrivastava et al. 1991] Shrivastava, S., Dixon, G. N., and Parrington, G. D., "An Overview of the Arjuna Distributed Programming System," *IEEE Software*, pp. 66–73 (January 1991).

[Singhal and Shivaratri 1994] Singhal, M., and Shivaratri, N. G., *Advanced Concepts in Operating Systems*, McGraw Hill, New York (1994).

[Sinha et al. 1991] Sinha, P. K., Ashihara, H., Shimizu, K., and Maekawa, M., "Flexible User-Definable Memory Coherence Scheme in Distributed Shared Memory of GALAXY," In: *Proceedings of the 2nd European Distributed Memory Computing Conference (EDMCC2)*, Springer-Verlag, New York, pp. 52–61 (April 1991).

[Smith 1982] Smith, A. J., "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, pp. 437–530, New York, NY (1982).

[Stumm and Zhou 1990] Stumm, M., and Zhou, S., "Algorithms Implementing Distributed Shared Memory," *IEEE Computer*, Vol. 23, No. 5, New York, NY, pp. 54–64 (1990).

[Tam and Hsu 1990] Tam, V., and Hsu, M., "Fast Recovery in Distributed Shared Virtual Memory Systems," In: *Proceedings of the 10th International Conference on Distributed Computing Systems*, IEEE, New York, pp. 38–45 (May–June 1990).

[Tam et al. 1990] Tam, M. C., Smith, J. M., and Farber, D. J., "A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems," *Operating Systems Review*, Vol. 24, pp. 40–67 (1990).

[Tartalja and Milutinovic 1996] Tartalja, I., and Milutinovic, V. (Eds.), *The Cache Coherence Problem in Shared-Memory Multiprocessors: Software Solutions*, IEEE Computer Society Press, Los Alamitos, CA (1996).

[Theel and Fleisch 1995] Theel, O. E., and Fleisch, B. D., "Design and Analysis of Highly Available and Scalable Coherence Protocols for Distributed Shared Memory Systems Using Stochastic Modeling," In: *Proceedings of the 24th Annual International Conference on Parallel Processing*, IEEE, New York (August 1995).

[Wu and Fuchs 1989] Wu, K. L., and Fuchs, W. K., "Recoverable Distributed Shared Virtual Memory: Memory Coherence and Storage Structures," In: *Proceedings of the 19th International Symposium on Fault-Tolerant Computing*, pp. 520–527 (June 1989).

[Yen et al. 1985] Yen, D. W. L., Yen, W. C., and Fu, K., "Data Coherence Problem in a Multicache System," *IEEE Transactions on Computers*, Vol. C-34, No. 1, pp. 56–65, New York, NY (1985).

[Zhou et al. 1990] Zhou, S., Stumm, M., and McInerney, T., "Extending Distributed Shared Memory to Heterogeneous Environments," In: *Proceedings of the 10th International Conference on Distributed Computing Systems*, IEEE, New York, pp. 30–37 (May–June 1990).

[Zhou et al. 1992] Zhou, S., Stumm, M., Li, K., and Wortman, D., "Heterogeneous Distributed Shared Memory," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 13, No. 5, New York, NY, pp. 540–554 (1992).

POINTERS TO BIBLIOGRAPHIES ON THE INTERNET

Bibliography containing references on *Distributed Shared Memory* can be found at:

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Parallel/dsm.html

Bibliography containing references on *Distributed Memory Systems* can be found at:

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Parallel/distmem.html

Bibliography containing references on *Cache Memories and Related Topics* can be found at:

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Misc/cache.html

Bibliography containing references on *Single Address Space Operating Systems (SASOS) and Related Topics* can be found at:

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Os/sasos.html

CHAPTER 6

Synchronization

6.1 INTRODUCTION

A distributed system consists of a collection of distinct processes that are spatially separated and run concurrently. In systems with multiple concurrent processes, it is economical to share the system resources (hardware or software) among the concurrently executing processes. In such a situation, sharing may be cooperative or competitive. That is, since the number of available resources in a computing system is restricted, one process must necessarily influence the action of other concurrently executing processes as it competes for resources. For example, for a resource (such as a tape drive) that cannot be used simultaneously by multiple processes, a process willing to use it must wait if another process is using it. At times, concurrent processes must cooperate either to achieve the desired performance of the computing system or due to the nature of the computation being performed. Typical examples of process cooperation involve two processes that bear a producer-consumer or client-server relationship to each other. For instance, a client process and a file server process must cooperate when performing file access operations. Both cooperative and competitive sharing require adherence to certain rules of behavior that guarantee that correct interaction occurs. The rules for enforcing correct interaction are implemented in the form of synchronization mechanisms. This chapter presents

synchronization mechanisms that are suitable for distributed systems. In particular, the following synchronization-related issues are described:

- Clock synchronization
- Event ordering
- Mutual exclusion
- Deadlock
- Election algorithms

6.2 CLOCK SYNCHRONIZATION

Every computer needs a timer mechanism (called a computer clock) to keep track of current time and also for various accounting purposes such as calculating the time spent by a process in CPU utilization, disk I/O, and so on, so that the corresponding user can be charged properly. In a distributed system, an application may have processes that concurrently run on multiple nodes of the system. For correct results, several such distributed applications require that the clocks of the nodes are synchronized with each other. For example, for a distributed on-line reservation system to be fair, the only remaining seat booked almost simultaneously from two different nodes should be offered to the client who booked first, even if the time difference between the two bookings is very small. It may not be possible to guarantee this if the clocks of the nodes of the system are not synchronized. In a distributed system, synchronized clocks also enable one to measure the duration of distributed activities that start on one node and terminate on another node, for instance, calculating the time taken to transmit a message from one node to another at any arbitrary time. It is difficult to get the correct result in this case if the clocks of the sender and receiver nodes are not synchronized. There are several other applications of synchronized clocks in distributed systems. Some good examples of such applications may be found in [Liskov 1993].

The discussion above shows that it is the job of a distributed operating system designer to devise and use suitable algorithms for properly synchronizing the clocks of a distributed system. This section presents a description of such algorithms. However, for a better understanding of these algorithms, we will first discuss how computer clocks are implemented and what are the main issues in synchronizing the clocks of a distributed system.

6.2.1 How Computer Clocks Are Implemented

A computer clock usually consists of three components—a quartz crystal that oscillates at a well-defined frequency, a *counter* register, and a *constant* register. The constant register is used to store a constant value that is decided based on the frequency of oscillation of the quartz crystal. The counter register is used to keep track of the oscillations of the quartz crystal. That is, the value in the counter register is decremented by 1 for each oscillation of the quartz crystal. When the value of the counter register becomes zero, an

interrupt is generated and its value is reinitialized to the value in the constant register. Each interrupt is called a *clock tick*.

To make the computer clock function as an ordinary clock used by us in our day-to-day life, the following things are done:

1. The value in the constant register is chosen so that 60 clock ticks occur in a second.
2. The computer clock is synchronized with real time (external clock). For this, two more values are stored in the system—a fixed starting date and time and the number of ticks. For example, in UNIX, time begins at 0000 on January 1, 1970. At the time of initial booting, the system asks the operator to enter the current date and time. The system converts the entered value to the number of ticks after the fixed starting date and time. At every clock tick, the interrupt service routine increments the value of the number of ticks to keep the clock running.

6.2.2 Drifting of Clocks

A clock always runs at a constant rate because its quartz crystal oscillates at a well-defined frequency. However, due to differences in the crystals, the rates at which two clocks run are normally different from each other. The difference in the oscillation period between two clocks might be extremely small, but the difference accumulated over many oscillations leads to an observable difference in the times of the two clocks, no matter how accurately they were initialized to the same value. Therefore, with the passage of time, a computer clock drifts from the real-time clock that was used for its initial setting. For clocks based on a quartz crystal, the drift rate is approximately 10^{-6}, giving a difference of 1 second every 1,000,000 seconds, or 11.6 days [Coulouris et al. 1994]. Hence a computer clock must be periodically resynchronized with the real-time clock to keep it nonfaulty. Even nonfaulty clocks do not always maintain perfect time. A clock is considered nonfaulty if there is a bound on the amount of drift from real time for any given finite time interval.

More precisely, let us suppose that when the real time is t, the time value of a clock p is $C_p(t)$. If all clocks in the world were perfectly synchronized, we would have $C_p(t) = t$ for all p and all t. That is, if C denotes the time value of a clock, in the ideal case dC/dt should be 1. Therefore, if the *maximum drift rate* allowable is ρ, a clock is said to be nonfaulty if the following condition holds for it:

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$

As shown in Figure 6.1, after synchronization with a perfect clock, slow and fast clocks drift in opposite directions from the perfect clock. This is because for slow clocks $dC/dt < 1$ and for fast clocks $dC/dt > 1$.

A distributed system consists of several nodes, each with its own clock, running at its own speed. Because of the nonzero drift rates of all clocks, the set of clocks of a distributed system do not remain well synchronized without some periodic resynchroniza-

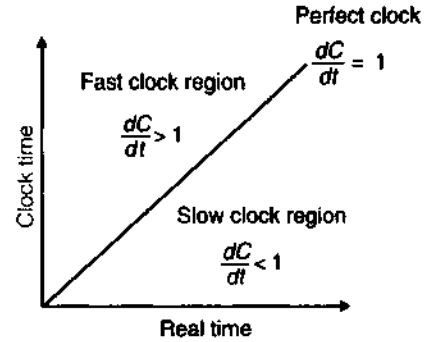

Fig. 6.1 Slow, perfect, and fast clocks.

tion. This means that the nodes of a distributed system must periodically resynchronize their local clocks to maintain a global time base across the entire system. Recall from Figure 6.1 that slow and fast clocks drift in opposite directions from the perfect clock. Therefore, of two clocks, if one is slow and one is fast, at a time Δt after they were synchronized, the maximum deviation between the time value of the two clocks will be $2\rho\Delta t$. Hence, to guarantee that no two clocks in a set of clocks ever differ by more than δ, the clocks in the set must be resynchronized periodically, with the time interval between two synchronizations being less than or equal to $\delta/2\rho$. Therefore, unlike a centralized system in which only the computer clock has to be synchronized with the real-time clock, a distributed system requires the following types of clock synchronization:

1. *Synchronization of the computer clocks with real-time (or external) clocks.* This type of synchronization is mainly required for real-time applications. That is, external clock synchronization allows the system to exchange information about the timing of events with other systems and users.

An external time source that is often used as a reference for synchronizing computer clocks with real time is the *Coordinated Universal Time* (*UTC*). The UTC is an international standard. Many standard bodies disseminate UTC signals by radio, telephone, and satellite. For instance, the WWV radio station in the United States and the Geostationary Operational Environmental Satellites (GEOS) are two such standard bodies. Commercial devices (called *time providers*) are available to receive and interpret these signals. Computers equipped with time provider devices can synchronize their clocks with these timing signals.

2. *Mutual (or internal) synchronization of the clocks of different nodes of the system.* This type of synchronization is mainly required for those applications that require a consistent view of time across all nodes of a distributed system as well as for the measurement of the duration of distributed activities that terminate on a node different from the one on which they start.

Note that externally synchronized clocks are also internally synchronized. However, the converse is not true because with the passage of time internally synchronized clocks may drift arbitrarily far from external time.

6.2.3 Clock Synchronization Issues

We have seen that no two clocks can be perfectly synchronized. Therefore, in practice, two clocks are said to be synchronized at a particular instance of time if the difference in time values of the two clocks is less than some specified constant δ. The difference in time values of two clocks is called *clock skew*. Therefore, a set of clocks are said to be synchronized if the clock skew of any two clocks in this set is less than δ.

Clock synchronization requires each node to read the other nodes' clock values. The actual mechanism used by a node to read other clocks differs from one algorithm to another. However, regardless of the actual reading mechanism, a node can obtain only an approximate view of its clock skew with respect to other nodes' clocks in the system. Errors occur mainly because of unpredictable communication delays during message passing used to deliver a clock signal or a clock message from one node to another. A minimum value of the unpredictable communication delays between two nodes can be computed by counting the time needed to prepare, transmit, and receive an empty message in the absence of transmission errors and any other system load. However, in general, it is rather impossible to calculate the upper bound of this value because it depends on the amount of communication and computation going on in parallel in the system, on the possibility that transmission errors will cause messages to be transmitted several times, and on other random events, such as page faults, process switches, or the establishment of new communication routes.

An important issue in clock synchronization is that time must never run backward because this could cause serious problems, such as the repetition of certain operations that may be hazardous in certain cases. Notice that during synchronization a fast clock has to be slowed down. However, if the time of a fast clock is readjusted to the actual time all at once, it may lead to running the time backward for that clock. Therefore, clock synchronization algorithms are normally designed to gradually introduce such a change in the fast running clock instead of readjusting it to the correct time all at once. One way to do this is to make the interrupt routine more intelligent. When an intelligent interrupt routine is instructed by the clock synchronization algorithm to slow down its clock, it readjusts the amount of time to be added to the clock time for each interrupt. For example, suppose that if 8 msec is added to the clock time on each interrupt in the normal situation, when slowing down, the interrupt routine only adds 7 msec on each interrupt until the correction has been made. Although not necessary, for smooth readjustment, the intelligent interrupt routine may also advance its clock forward, if it is found to be slow, by adding 9 msec on each interrupt, instead of readjusting it to the correct time all at once.

6.2.4 Clock Synchronization Algorithms

Clock synchronization algorithms may be broadly classified as centralized and distributed.

Centralized Algorithms

In centralized clock synchronization algorithms one node has a real-time receiver. This node is usually called the *time server node*, and the clock time of this node is regarded as correct and used as the reference time. The goal of the algorithm is to keep the clocks of all other nodes synchronized with the clock time of the time server node. Depending on the role of the time server node, centralized clock synchronization algorithms are again of two types—passive time server and active time server.

Passive Time Server Centralized Algorithm. In this method, each node periodically (with the interval between two periods being less than or equal to $\delta/2\rho$) sends a message ("*time* = *?*") to the time server. When the time server receives the message, it quickly responds with a message ("*time* = T"), where T is the current time in the clock of the time server node. Let us assume that when the client node sends the "*time* = *?*" message, its clock time is T_0, and when it receives the "*time* = T" message, its clock time is T_1. Since T_0 and T_1 are measured using the same clock, in the absence of any other information, the best estimate of the time required for the propagation of the message "*time* = T" from the time server node to the client's node is $(T_1 - T_0)/2$. Therefore, when the reply is received at the client's node, its clock is readjusted to $T+(T_1 - T_0)/2$.

Since there may be unpredictable variation in the message propagation time between two nodes, $(T_1 - T_0)/2$ is not a very good estimate of the time to be added to T for calculating the current time of the client's node clock. Several proposals have been made to improve this estimated value. Two such methods are described below. The first one assumes the availability of some additional information and the second one assumes that no additional information is available:

1. In this method, it is assumed that the approximate time taken by the time server to handle the interrupt and process a "*time* = *?*" request message is known. Let this time be equal to I. Then a better estimate of the time taken for propagation of the message "*time* = T" from the time server node to the client's node would be $(T_1 - T_0 - I)/2$. Therefore, in this method, when the reply is received at the client's node, its clock is readjusted to $T + (T_1 - T_0 - I)/2$.

2. This method was proposed by Cristian [1989]. In this method, several measurements of $T_1 - T_0$ are made, and those measurements for which $T_1 - T_0$ exceeds some threshold value are considered to be unreliable and discarded. The average of the remaining measurements is then calculated, and half of the calculated value is used as the value to be added to T. Alternatively, the measurement for which the value of $T_1 - T_0$ is minimum is considered to be the most accurate one, and half of this value is used as the value to be added to T. One limitation of this approach is the need to restrict the number of measurements for estimating the value to be added to T, since these are directly related to the message traffic generated and the overhead imposed by the algorithm.

Active Time Server Centralized Algorithm. In the passive time server approach, the time server only responds to requests for time from other nodes. On the other hand, in the active time server approach, the time server periodically broadcasts its clock time ("*time* = T"). The other nodes receive the broadcast message and use the clock time in

the message for correcting their own clocks. Each node has a priori knowledge of the approximate time (T_a) required for the propagation of the message "*time* = *T*" from the time sever node to its own node. Therefore, when the broadcast message is received at a node, the node's clock is readjusted to the time $T + T_a$. A major drawback of this method is that it is not fault tolerant. If the broadcast message reaches too late at a node due to some communication fault, the clock of that node will be readjusted to an incorrect value. Another drawback of this approach is that it requires broadcast facility to be supported by the network.

Another active time server algorithm that overcomes the drawbacks of the above algorithm is the *Berkeley algorithm*. It was proposed by Gusella and Zatti [1989] for internal synchronization of clocks of a group of computers running the Berkeley UNIX. In this algorithm, the time server periodically sends a message ("*time* = *?*") to all the computers in the group. On receiving this message, each computer sends back its clock value to the time server. The time server has a priori knowledge of the approximate time required for the propagation of a message from each node to its own node. Based on this knowledge, it first readjusts the clock values of the reply messages. It then takes a *fault-tolerant average* of the clock values of all the computers (including its own). To take the fault-tolerant average, the time server chooses a subset of all clock values that do not differ from one another by more than a specified amount, and the average is taken only for the clock values in this subset. This approach eliminates readings from unreliable clocks whose clock values could have a significant adverse effect if an ordinary average was taken.

The calculated average is the current time to which all the clocks should be readjusted. The time server readjusts its own clock to this value. However, instead of sending the calculated current time back to the other computers, the time server sends the amount by which each individual computer's clock requires adjustment. This can be a positive or a negative value and is calculated based on the knowledge the time server has about the approximate time required for the propagation of a message from each node to its own node.

Centralized clock synchronization algorithms suffer from two major drawbacks:

1. They are subject to single-point failure. If the time server node fails, the clock synchronization operation cannot be performed. This makes the system unreliable. Ideally, a distributed system should be more reliable than its individual nodes. If one goes down, the rest should continue to function correctly.
2. From a scalability point of view it is generally not acceptable to get all the time requests serviced by a single time server. In a large system, such a solution puts a heavy burden on that one process.

Distributed algorithms overcome these drawbacks.

Distributed Algorithms

Recall that externally synchronized clocks are also internally synchronized. That is, if each node's clock is independently synchronized with real time, all the clocks of the system remain mutually synchronized. Therefore, a simple method for clock synchroniza-

tion may be to equip each node of the system with a real-time receiver so that each node's clock can be independently synchronized with real time. Multiple real-time clocks (one for each node) are normally used for this purpose.

Theoretically, internal synchronization of clocks is not required in this approach. However, in practice, due to the inherent inaccuracy of real-time clocks, different real-time clocks produce different time. Therefore, internal synchronization is normally performed for better accuracy. One of the following two approaches is usually used for internal synchronization in this case.

Global Averaging Distributed Algorithms. In this approach, the clock process at each node broadcasts its local clock time in the form of a special "resync" message when its local time equals $T_0 + iR$ for some integer i, where T_0 is a fixed time in the past agreed upon by all nodes and R is a system parameter that depends on such factors as the total number of nodes in the system, the maximum allowable drift rate, and so on. That is, a resync message is broadcast from each node at the beginning of every fixed-length resynchronization interval. However, since the clocks of different nodes run at slightly different rates, these broadcasts will not happen simultaneously from all nodes.

After broadcasting the clock value, the clock process of a node waits for time T, where T is a parameter to be determined by the algorithm. During this waiting period, the clock process collects the resync messages broadcast by other nodes. For each resync message, the clock process records the time, according to its own clock, when the message was received. At the end of the waiting period, the clock process estimates the skew of its clock with respect to each of the other nodes on the basis of the times at which it received resync messages. It then computes a fault-tolerant average of the estimated skews and uses it to correct the local clock before the start of the next resynchronization interval.

The global averaging algorithms differ mainly in the manner in which the fault-tolerant average of the estimated skews is calculated. Two commonly used algorithms are described here:

1. The simplest algorithm is to take the average of the estimated skews and use it as the correction for the local clock. However, to limit the impact of faulty clocks on the average value, the estimated skew with respect to each node is compared against a threshold, and skews greater than the threshold are set to zero before computing the average of the estimated skews.

2. In another algorithm, each node limits the impact of faulty clocks by first discarding the m highest and m lowest estimated skews and then calculating the average of the remaining skews, which is then used as the correction for the local clock. The value of m is usually decided based on the total number of clocks (nodes).

Localized Averaging Distributed Algorithms. The global averaging algorithms do not scale well because they require the network to support broadcast facility and also because of the large amount of message traffic generated. Therefore, they are suitable for small networks, especially for those that have fully connected topology (in which each node has a direct communication link to every other node). The localized averaging algorithms attempt to overcome these drawbacks of the global averaging algorithms. In

this approach, the nodes of a distributed system are logically arranged in some kind of pattern, such as a ring or a grid. Periodically, each node exchanges its clock time with its neighbors in the ring, grid, or other structure and then sets its clock time to the average of its own clock time and the clock times of its neighbors.

6.2.5 Case Study: Distributed Time Service

Two popular services for synchronizing clocks and for providing timing information over a wide variety of interconnected networks are the Distributed Time Service (DTS) and the Network Time Protocol (NTP). DTS is a component of DCE (Distributed Computing Environment) that is used to synchronize clocks of a network of computers running DCE, and NTP is used in the Internet for clock synchronization. DTS is briefly described below as a case study of clock synchronization. Details of NTP can be found in [Mills 1991].

In a DCE system, each node is configured as either a *DTS client* or a *DTS server*. On each DTS client node runs a daemon process called a *DTS clerk*. To synchronize its local clock, each DTS clerk makes requests to the DTS servers on the same LAN for timing information. The DTS servers provide timing information to DTS clerks or to other DTS servers upon request. To make them publicly known, each DTS server exports its name to a LAN profile.

DTS does not define time as a single value. Instead, time is expressed as an interval containing the correct time. By using intervals instead of values, DTS provides the users with a clear idea of how far off the clock might be from reference time.

A DTS clerk synchronizes its local clock in the following manner. It keeps track of the drift rate of its local clock, and when it discovers that the time error of the local clock has exceeded the allowable limit, it initiates resynchronization by doing an RPC with all the DTS servers on its LAN requesting for the time. Each DTS server that receives this message returns a reply containing a time interval based on the server's own clock. From the received replies, the DTS clerk computes its new value of time in the following manner (see Fig. 6.2 for an example). At first, time intervals that do not intersect with the majority of time intervals are considered to be faulty and discarded. For instance, in Figure 6.2, the value supplied by DTS server 3 is discarded. Then the largest intersection falling within the remaining intervals is computed. The DTS clerk then resets its clock value to the midpoint of this interval. However, instead of resetting the clock to the calculated value all at once, an intelligent interrupt routine is used to gradually introduce such a change in the clock time.

Note that due to the use of the method of intersection for computing new clock value, it is recommended that each LAN in a DCE system should have at least three DTS servers to provide time information.

In addition to DTS clerks synchronizing their clocks with DTS servers, the DTS servers of a LAN also communicate among themselves periodically to keep their clocks mutually synchronized. They also use the algorithm of Figure 6.2 to compute the new clock value.

So far we have seen how clocks of nodes belonging to the same LAN are synchronized. However, a DCE system may have several interconnected LANs. In this case, a need arises to synchronize the clocks of all nodes in the network. For this, one DTS

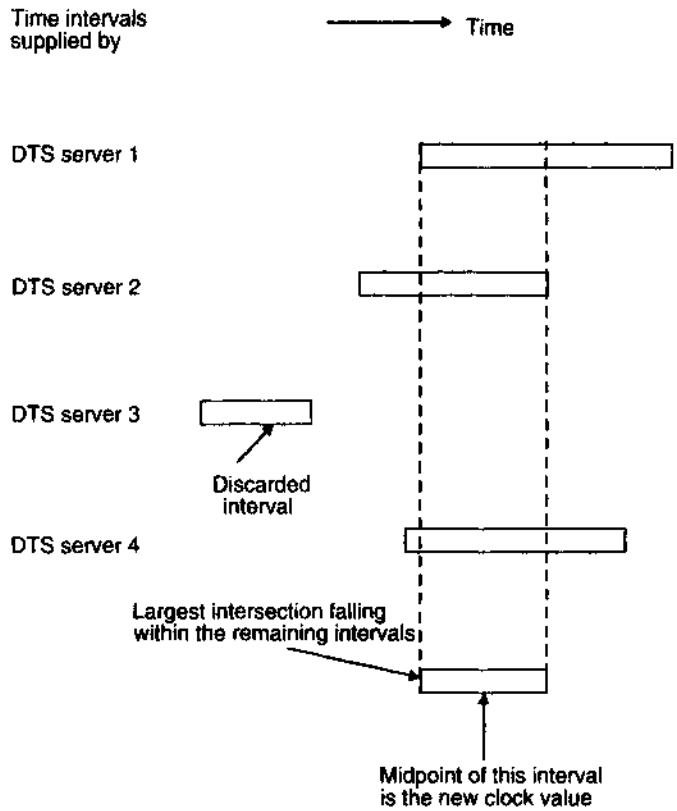

Fig. 6.2 Computation of new clock value in DTS from obtained time intervals.

server of each LAN is designated a *global server*. Although not necessary for external synchronization, it is recommended that each global server be equipped with a time provider device to receive UTC signals. The global servers of all LANs communicate among themselves periodically to keep their clocks mutually synchronized. Since the global server of a LAN is also a DTS server, its clock value is automatically used to synchronize the clocks of other nodes in the LAN. In this manner, DTS synchronizes the clocks of all nodes in the network.

DTS is transparent to DCE users in the sense that users cannot access DTS directly. However, DTS application programming interface (API) provides a rich set of library procedures to allow DTS applications to perform time-related activities to control their executions. In particular, there are library procedures to get the current time, to convert between binary and ASCII representations of time, to manipulate binary time information, to compare two times, to perform arithmetic operations on times, to get time zone information, and so on. In addition, there is an administrative interface to DTS that allows a system administrator to perform administrative operations such as configuring and dynamically reconfiguring the number of DTS clients and DTS servers on a LAN,

changing a DTS server into a global server when the global server of a LAN fails, and setting the maximum inaccuracy and error tolerance to decide how frequently resynchronization should take place.

6.3 EVENT ORDERING

Keeping the clocks in a distributed system synchronized to within 5 or 10 msec is an expensive and nontrivial task. Lamport [1978] observed that for most applications it is not necessary to keep the clocks in a distributed system synchronized. Rather, it is sufficient to ensure that all events that occur in a distributed system be totally ordered in a manner that is consistent with an observed behavior.

For partial ordering of events, Lamport defined a new relation called *happened-before* and introduced the concept of logical clocks for ordering of events based on the happened-before relation. He then gave a distributed algorithm extending his idea of partial ordering to a consistent total ordering of all the events in a distributed system. His idea is presented below.

6.3.1 Happened-Before Relation

The happened-before relation (denoted by $\rightarrow$) on a set of events satisfies the following conditions:

1. If a and b are events in the same process and a occurs before b, then $a \rightarrow b$.
2. If a is the event of sending a message by one process and b is the event of the receipt of the same message by another process, then $a \rightarrow b$. This condition holds by the law of causality because a receiver cannot receive a message until the sender sends it, and the time taken to propagate a message from its sender to its receiver is always positive.
3. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. That is, happened-before is a transitive relation.

Notice that in a physically meaningful system, an event cannot happen before itself, that is, $a \rightarrow a$ is not true for any event a. This implies that happened-before is an irreflexive partial ordering on the set of all events in the system.

In terms of the happened-before relation, two events a and b are said to be *concurrent* if they are not related by the happened-before relation. That is, neither $a \rightarrow b$ nor $b \rightarrow a$ is true. This is possible if the two events occur in different processes that do not exchange messages either directly or indirectly via other processes. Notice that this definition of concurrency simply means that nothing can be said about when the two events happened or which one happened first. That is, two events are concurrent if neither can causally affect the other. Due to this reason, the happened-before relation is sometimes also known as the relation of *causal ordering*.

A space-time diagram (such as the one shown in Fig. 6.3) is often used to illustrate the concepts of the happened-before relation and concurrent events. In this diagram, each

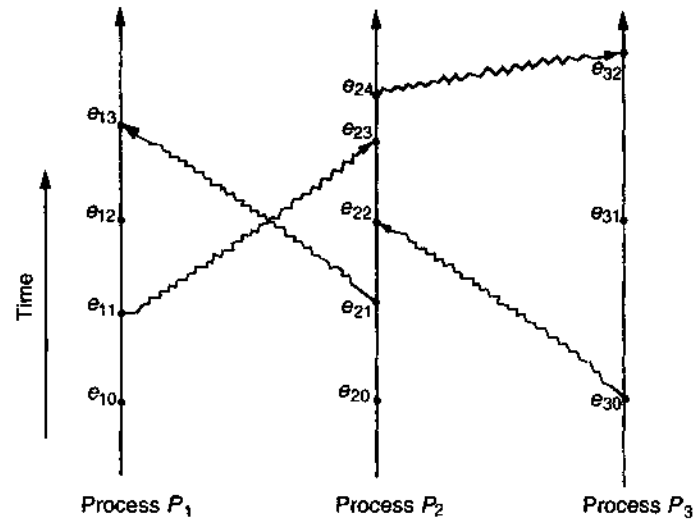

Fig. 6.3 Space-time diagram for three processes.

vertical line denotes a process, each dot on a vertical line denotes an event in the corresponding process, and each wavy line denotes a message transfer from one process to another in the direction of the arrow.

From this space-time diagram it is easy to see that for two events a and b, $a \rightarrow b$ is true if and only if there exists a path from a to b by moving forward in time along process and message lines in the direction of the arrows. For example, some of the events of Figure 6.3 that are related by the happened-before relation are

$$e_{10} \rightarrow e_{11} \qquad e_{20} \rightarrow e_{24} \qquad e_{11} \rightarrow e_{23} \qquad e_{21} \rightarrow e_{13}$$
$$e_{30} \rightarrow e_{24} \quad (\text{since } e_{30} \rightarrow e_{22} \text{ and } e_{22} \rightarrow e_{24})$$
$$e_{11} \rightarrow e_{32} \quad (\text{since } e_{11} \rightarrow e_{23},\ e_{23} \rightarrow e_{24}, \text{ and } e_{24} \rightarrow e_{32})$$

On the other hand, two events a and b are concurrent if and only if no path exists either from a to b or from b to a. For example, some of the concurrent events of Figure 6.3 are

e_{12} and e_{20} $\quad e_{21}$ and e_{30} $\quad e_{10}$ and e_{30} $\quad e_{11}$ and e_{31} $\quad e_{12}$ and e_{32} $\quad e_{13}$ and e_{22}

6.3.2 Logical Clocks Concept

To determine that an event a happened before an event b, either a common clock or a set of perfectly synchronized clocks is needed. We have seen that neither of these is available in a distributed system. Therefore, in a distributed system the happened-before relation must be defined without the use of globally synchronized physical clocks.

Lamport [1978] provided a solution for this problem by introducing the concept of logical clocks.

The logical clocks concept is a way to associate a timestamp (which may be simply a number independent of any clock time) with each system event so that events that are related to each other by the happened-before relation (directly or indirectly) can be properly ordered in that sequence. Under this concept, each process P_i has a clock C_i associated with it that assigns a number $C_i(a)$ to any event a in that process. The clock of each process is called a logical clock because no assumption is made about the relation of the numbers $C_i(a)$ to physical time. In fact, the logical clocks may be implemented by counters with no actual timing mechanism. With each process having its own clock, the entire system of clocks is represented by the function C, which assigns to any event b the number $C(b)$, where $C(b) = C_j(b)$ if b is an event in process P_j.

The logical clocks of a system can be considered to be correct if the events of the system that are related to each other by the happened-before relation can be properly ordered using these clocks. Therefore, the timestamps assigned to the events by the system of logical clocks must satisfy the following *clock condition:*

For any two events a and b, if $a \rightarrow b$, then $C(a) < C(b)$.

Note that we cannot expect the converse condition to hold as well, since that would imply that any two concurrent events must occur at the same time, which is not necessarily true for all concurrent events.

6.3.3 Implementation of Logical Clocks

From the definition of the happened-before relation, it follows that the clock condition mentioned above is satisfied if the following conditions hold:

C1: If a and b are two events within the same process P_i and a occurs before b, then $C_i(a) < C_i(b)$.

C2: If a is the sending of a message by process P_i and b is the receipt of that message by process P_j, then $C_i(a) < C_j(b)$.

In addition to these conditions, which are necessary to satisfy the clock condition, the following condition is necessary for the correct functioning of the system:

C3: A clock C_i associated with a process P_i must always go forward, never backward. That is, corrections to time of a logical clock must always be made by adding a positive value to the clock, never by subtracting value.

Obviously, any algorithm used for implementing a set of logical clocks must satisfy all these three conditions. The algorithm proposed by Lamport is given below.

To meet conditions C1, C2, and C3, Lamport's algorithm uses the following implementation rules:

IR1: Each process P_i increments C_i between any two successive events.

IR2: If event a is the sending of a message m by process P_i, the message m contains a timestamp $T_m = C_i(a)$, and upon receiving the message m a process P_j sets C_j greater than or equal to its present value but greater than T_m.

Rule IR1 ensures that condition C1 is satisfied and rule IR2 ensures that condition C2 is satisfied. Both IR1 and IR2 ensure that condition C3 is also satisfied. Hence the simple implementation rules IR1 and IR2 guarantee a correct system of logical clocks.

The implementation of logical clocks can best be illustrated with an example. How a system of logical clocks can be implemented either by using counters with no actual timing mechanism or by using physical clocks is shown below.

Implementation of Logical Clocks by Using Counters

As shown in Figure 6.4, two processes P_1 and P_2 each have a counter C_1 and C_2, respectively. The counters act as logical clocks. At the beginning, the counters are initialized to zero and a process increments its counter by 1 whenever an event occurs in that process. If the event is sending of a message (e.g., events e_{04} and e_{14}), the process includes the incremented value of the counter in the message. On the other hand, if the event is receiving of a message (e.g., events e_{13} and e_{08}), instead of simply incrementing the counter by 1, a check is made to see if the incremented counter value is less than or equal to the timestamp in the received message. If so, the counter value is corrected and

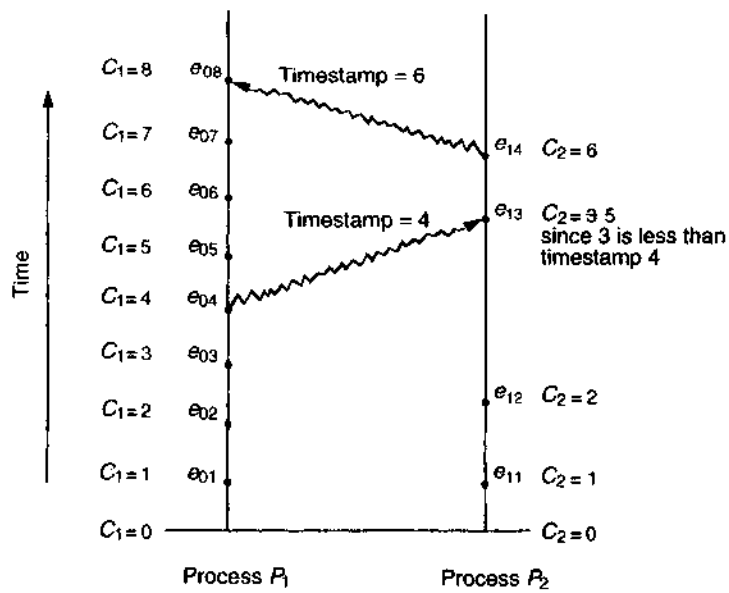

Fig. 6.4 Example illustrating the implementation of logical clocks by using counters.

set to 1 plus the timestamp in the received message (e.g., in event e_{13}). If not, the counter value is left as it is (e.g., in event e_{08}).

Implementation of Logical Clocks by Using Physical Clocks

The implementation of the example of Figure 6.4 by using physical clocks instead of counters is shown in Figure 6.5. In this case, each process has a physical clock associated with it. Each clock runs at a constant rate. However, the rates at which different clocks run are different. For instance, in the example of Figure 6.5, when the clock of process P_1 has ticked 10 times, the clock of process P_2 has ticked only 8 times.

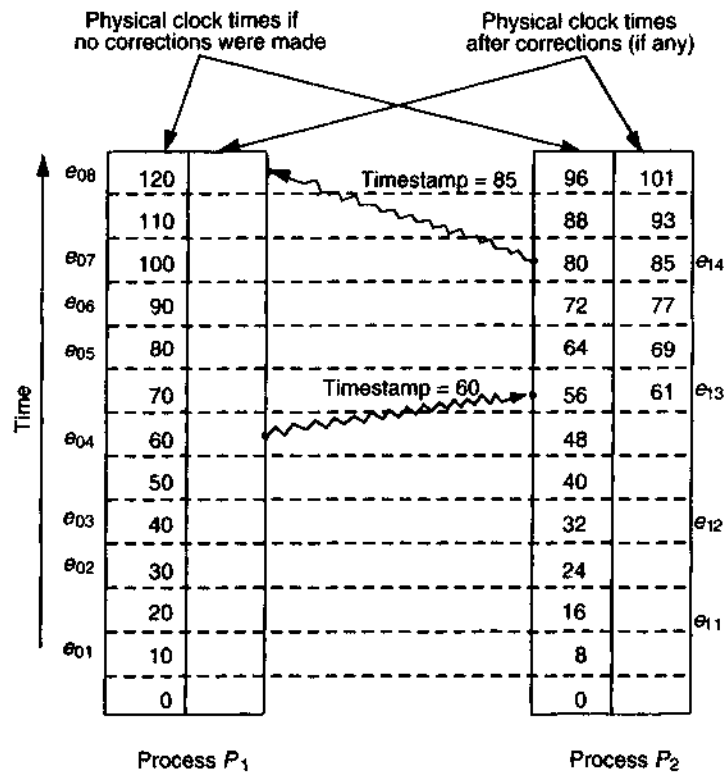

Fig. 6.5 Example illustrating the implementation of logical clocks by using physical clocks.

To satisfy condition C1, the only requirement is that the physical clock of a process must tick at least once between any two events in that process. This is usually not a problem because a computer clock is normally designed to click several times between two events that happen in quick succession. To satisfy condition C2, for a

message-sending event (e.g., events e_{04} and e_{14}), the process sending the message includes its current physical time in the message. And for a message-receiving event (e.g., events e_{13} and e_{08}), a check is made to see if the current time in the receiver's clock is less than or equal to the time included in the message. If so, the receiver's physical clock is corrected by fast forwarding its clock to be 1 more than the time included in the message (e.g., in event e_{13}). If not, the receiver's clock is left as it is (e.g., in event e_{08}).

6.3.4 Total Ordering of Events

We have seen how a system of clocks satisfying the clock condition can be used to order the events of a system based on the happened-before relationship among the events. We simply need to order the events by the times at which they occur. However, recall that the happened-before relation is only a partial ordering on the set of all events in the system. With this event-ordering scheme, it is possible that two events a and b that are not related by the happened-before relation (either directly or indirectly) may have the same timestamps associated with them. For instance, if events a and b happen respectively in processes P_1 and P_2, when the clocks of both processes show exactly the same time (say 100), both events will have a timestamp of 100. In this situation, nothing can be said about the order of the two events. Therefore, for total ordering on the set of all system events, an additional requirement is desirable: No two events ever occur at exactly the same time. To fulfill this requirement, Lamport proposed the use of any arbitrary total ordering of the processes. For example, process identity numbers may be used to break ties and to create a total ordering of events. For instance, in the situation described above, the timestamps associated with events a and b will be 100.001 and 100.002, respectively, where the process identity numbers of processes P_1 and P_2 are 001 and 002, respectively. Using this method, we now have a way to assign a unique timestamp to each event in a distributed system to provide a total ordering of all events in the system.

6.4 MUTUAL EXCLUSION

There are several resources in a system that must not be used simultaneously by multiple processes if program operation is to be correct. For example, a file must not be simultaneously updated by multiple processes. Similarly, use of unit record peripherals such as tape drives or printers must be restricted to a single process at a time. Therefore, exclusive access to such a shared resource by a process must be ensured. This exclusiveness of access is called *mutual exclusion* between processes. The sections of a program that need exclusive access to shared resources are referred to as *critical sections*. For mutual exclusion, means are introduced to prevent processes from executing concurrently within their associated critical sections.

An algorithm for implementing mutual exclusion must satisfy the following requirements:

1. *Mutual exclusion.* Given a shared resource accessed by multiple concurrent processes, at any time only one process should access the resource. That is, a process that has been granted the resource must release it before it can be granted to another process.
2. *No starvation.* If every process that is granted the resource eventually releases it, every request must be eventually granted.

In single-processor systems, mutual exclusion is implemented using semaphores, monitors, and similar constructs. The three basic approaches used by different algorithms for implementing mutual exclusion in distributed systems are described below. Interested readers who want to explore further on this topic may refer to [Agarwal and Abbadi 1991, Bulgannawar and Vaidya 1995, Raynal 1991, Sanders 1987, Suzuki and Kasami 1985]. To simplify our description, we assume that each process resides at a different node.

6.4.1 Centralized Approach

In this approach, one of the processes in the system is elected as the coordinator (algorithms for electing a coordinator are described later in this chapter) and coordinates the entry to the critical sections. Each process that wants to enter a critical section must first seek permission from the coordinator. If no other process is currently in that critical section, the coordinator can immediately grant permission to the requesting process. However, if two or more processes concurrently ask for permission to enter the same critical section, the coordinator grants permission to only one process at a time in accordance with some scheduling algorithm. After executing a critical section, when a process exits the critical section, it must notify the coordinator so that the coordinator can grant permission to another process (if any) that has also asked for permission to enter the same critical section.

An algorithm for mutual exclusion that uses the centralized approach is described here with the help of an example. As shown in Figure 6.6, let us suppose that there is a coordinator process (P_c) and three other processes P_1, P_2, and P_3 in the system. Also assume that the requests are granted in the first-come, first-served order for which the coordinator maintains a request queue. Suppose P_1 wants to enter a critical section for which it sends a *request* message to P_c. On receiving the request message, P_c checks to see whether some other process is currently in that critical section. Since no other process is in the critical section, P_c immediately sends back a *reply* message granting permission to P_1. When the reply arrives, P_1 enters the critical section.

Now suppose that while P_1 is in the critical section P_2 asks for permission to enter the same critical section by sending a request message to P_c. Since P_1 is already in the critical section, P_2 cannot be granted permission. The exact method used to deny permission varies from one algorithm to another. For our algorithm, let us assume that the coordinator does not return any reply and the process that made the request remains blocked until it receives the reply from the coordinator. Therefore, P_c does not send a reply to P_2 immediately and enters its request in the request queue.

Again suppose that while P_1 is still in the critical section P_3 also sends a request message to P_c asking for permission to enter the same critical section. Obviously, P_3

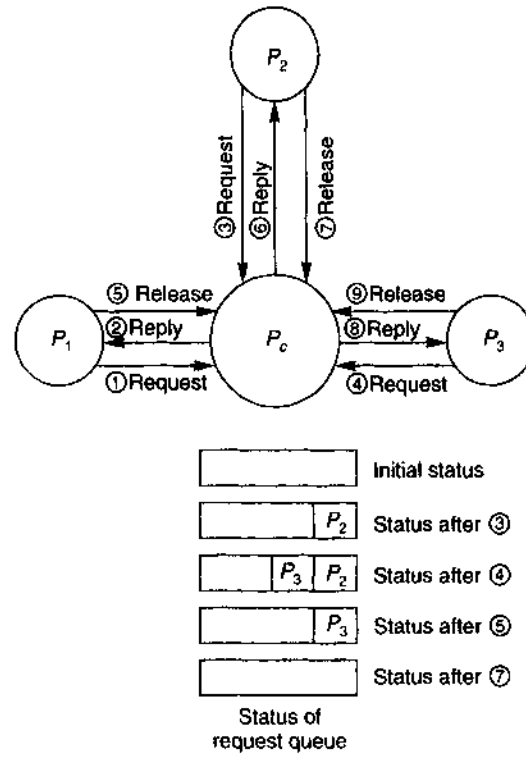

Fig. 6.6 Example illustrating the centralized approach for mutual exclusion.

cannot be granted permission, so no reply is sent immediately to P_3 by P_c, and its request is queued in the request queue.

Now suppose P_1 exits the critical section and sends a *release* message to P_c releasing its exclusive access to the critical section. On receiving the release message, P_c takes the first request from the queue of deferred requests and sends a reply message to the corresponding process, granting it permission to enter the critical section. Therefore, in this case, P_c sends a reply message to P_2.

On receiving the reply message, P_2 enters the critical section, and when it exits the critical section, it sends a release message to P_c. Again P_c takes the first request from the request queue (in this case request of P_3) and sends a reply message to the corresponding process (P_3). On receiving the reply message, P_3 enters the critical section, and when it exits the critical section, it sends a release message to P_c. Now since there are no more requests, P_c keeps waiting for the next request message.

This algorithm ensures mutual exclusion because, at a time, the coordinator allows only one process to enter a critical section. The algorithm also ensures that no starvation will occur because of the use of first-come, first-served scheduling policy. The main advantages of this algorithm is that it is simple to implement and requires only three messages per critical section entry: a request, a reply, and a release. However, it suffers from the usual drawbacks of centralized schemes. That is, a single coordinator is subject

to a single point of failure and can become a performance bottleneck in a large system. Furthermore, for failure handling, means must be provided to detect a failure of the coordinator, to elect a unique new coordinator, and to reconstruct its request queue before the computation can be resumed.

6.4.2 Distributed Approach

In the distributed approach, the decision making for mutual exclusion is distributed across the entire system. That is, all processes that want to enter the same critical section cooperate with each other before reaching a decision on which process will enter the critical section next. The first such algorithm was presented by Lamport [1978] based on his event-ordering scheme described in Section 6.3. Later, Ricart and Agrawala [1981] proposed a more efficient algorithm that also requires there be a total ordering of all events in the system. As an example of a distributed algorithm for mutual exclusion, Ricart and Agrawala's algorithm is described below. In the following description we assume that Lamport's event-ordering scheme is used to generate a unique timestamp for each event in the system.

When a process wants to enter a critical section, it sends a request message to all other processes. The message contains the following information:

1. The process identifier of the process
2. The name of the critical section that the process wants to enter
3. A unique timestamp generated by the process for the request message

On receiving a request message, a process either immediately sends back a reply message to the sender or defers sending a reply based on the following rules:

1. If the receiver process is itself currently executing in the critical section, it simply queues the request message and defers sending a reply.
2. If the receiver process is currently not executing in the critical section but is waiting for its turn to enter the critical section, it compares the timestamp in the received request message with the timestamp in its own request message that it has sent to other processes. If the timestamp of the received request message is lower, it means that the sender process made a request before the receiver process to enter the critical section. Therefore, the receiver process immediately sends back a reply message to the sender. On the other hand, if the receiver process's own request message has a lower timestamp, the receiver queues the received request message and defers sending a reply message.
3. If the receiver process neither is in the critical section nor is waiting for its turn to enter the critical section, it immediately sends back a reply message.

A process that sends out a request message keeps waiting for reply messages from other processes. It enters the critical section as soon as it has received reply messages from all processes. After it finishes executing in the critical section, it sends reply messages to all processes in its queue and deletes them from its queue.

To illustrate how the algorithm works, let us consider the example of Figure 6.7. There are four processes P_1, P_2, P_3, and P_4. While process P_4 is in a critical section, processes P_1 and P_2 want to enter the same critical section. To get permission from other processes, processes P_1 and P_2 send request messages with timestamps 6 and 4 respectively to other processes (Fig. 6.7(a)).

Now let us consider the situation in Figure 6.7(b). Since process P_4 is already in the critical section, it defers sending a reply message to P_1 and P_2 and enters them in its queue. Process P_3 is currently not interested in the critical section, so it sends a reply

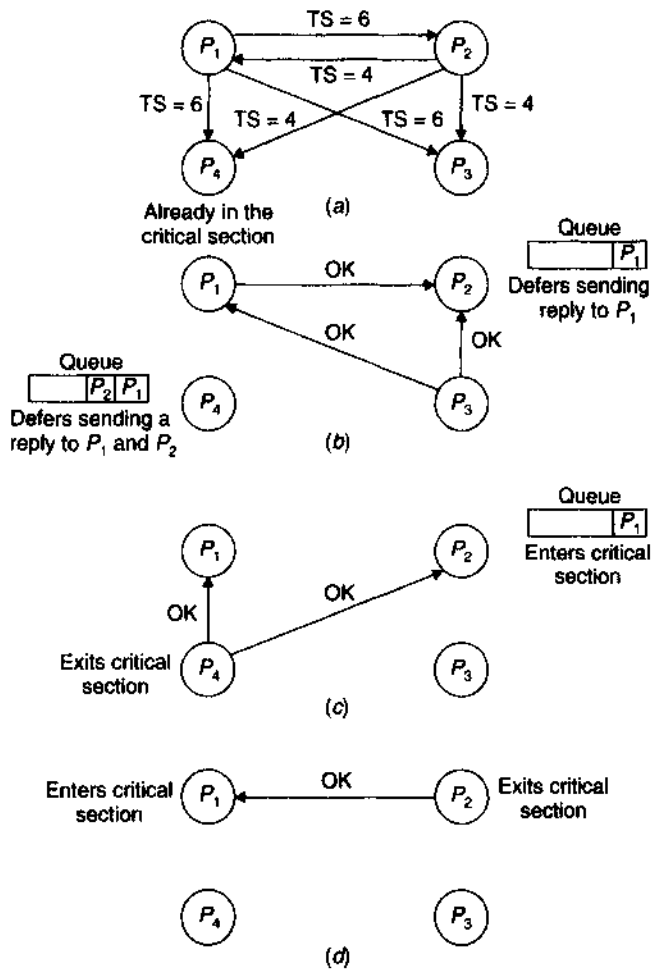

Fig. 6.7 Example illustrating the distributed algorithm for mutual exclusion: (a) status when processes P_1 and P_2 send request messages to other processes while process P_4 is already in the critical section; (b) status while process P_4 is still in critical section; (c) status after process P_4 exits critical section; (d) status after process P_2 exits critical section.

message to both P_1 and P_2. Process P_2 defers sending a reply message to P_1 and enters P_1 in its queue because the timestamp (4) in its own request message is less than the timestamp (6) in P_1's request message. On the other hand, P_1 immediately replies to P_2 because the timestamp (6) in its request message is found to be greater than the timestamp (4) of P_2's request message.

Next consider the situation in Figure 6.7(*c*). When process P_4 exits the critical section, it sends a reply message to all processes in its queue (in this case to processes P_1 and P_2) and deletes them from its queue. Now since process P_2 has received a reply message from all other processes (P_1, P_3, and P_4), it enters the critical section. However, process P_1 continues to wait since it has not yet received a reply message from process P_2.

Finally, when process P_2 exits the critical section, it sends a reply message to P_1 (Fig. 6.7(*d*)). Now since process P_1 has received a reply message from all other processes, it enters the critical section.

The algorithm guarantees mutual exclusion because a process can enter its critical section only after getting permission from all other processes, and in the case of a conflict only one of the conflicting processes can get permission from all other processes. The algorithm also ensures freedom from starvation since entry to the critical section is scheduled according to the timestamp ordering. It has also been proved by Ricart and Agrawala [1981] that the algorithm is free from deadlock. Furthermore, if there are n processes, the algorithm requires $n-1$ request messages and $n-1$ reply messages, giving a total of $2(n-1)$ messages per critical section entry. However, this algorithm suffers from the following drawbacks because of the requirement that all processes must participate in a critical section entry request by any process:

1. In a system having n processes, the algorithm is liable to n points of failure because if one of the processes fails, the entire scheme collapses. This is because the failed process will not reply to request messages that will be falsely interpreted as denial of permission by the requesting processes, causing all the requesting processes to wait indefinitely.

Tanenbaum [1995] proposed a simple modification to the algorithm to solve this problem. In the modified algorithm, instead of remaining silent by deferring the sending of the reply message in cases when permission cannot be granted immediately, the receiver sends a "permission denied" reply message to the requesting process and then later sends an OK message when the permission can be granted. Therefore, a reply message (either "permission denied" or OK) is immediately sent to the requesting process in any case. If the requesting process does not receive a reply from a process within a fixed timeout period, it either keeps trying until the process replies or concludes that the process has crashed. When the requesting process receives a "permission denied" reply message from one or more of the processes, it blocks until an OK message is received from all of them.

2. The algorithm requires that each process know the identity of all the processes participating in the mutual-exclusion algorithm. This requirement makes implementation of the algorithm complex because each process of a group needs to dynamically keep track

of the processes entering or leaving the group. That is, when a process joins a group, it must receive the names of all the other processes in the group, and the name of the new process must be distributed to all the other processes in the group. Similarly, when a process leaves the group or crashes, all members of that group must be informed so that they can delete it from their membership list. Updating of the membership list is particularly difficult when request and reply messages are already being exchanged among the processes of the group. Therefore, the algorithm is suitable only for groups whose member processes are fixed and do not change dynamically.

3. In this algorithm, a process willing to enter a critical section can do so only after communicating with all other processes and getting permission from them. Therefore, assuming that the network can handle only one message at a time, the waiting time from the moment the process makes a request to enter a critical region until it actually enters the critical section is the time for exchanging $2(n-1)$ messages in a system having n processes. This waiting time may be large if there are too many processes in the system. Therefore, the algorithm is suitable only for a small group of cooperating processes.

Some improvements to this algorithm have been proposed in the literature. For instance, a simple improvement is possible by using the idea of majority consensus rather than the consensus of all other processes for critical section entry [Tanenbaum 1995]. That is, in an algorithm that uses the idea of majority consensus, a process can enter a critical section as soon as it has collected permission from a majority of the other processes, rather than from all of them. Note that in this algorithm a process can grant permission for a critical section entry to only a single process at a time. Two other possible improvements to the algorithm can be found in [Carvalho and Roucairol 1983, Maekawa et al. 1987].

6.4.3 Token-Passing Approach

In this method, mutual exclusion is achieved by using a single token that is circulated among the processes in the system. A *token* is a special type of message that entitles its holder to enter a critical section. For fairness, the processes in the system are logically organized in a ring structure, and the token is circulated from one process to another around the ring always in the same direction (clockwise or anticlockwise).

The algorithm works as follows. When a process receives the token, it checks if it wants to enter a critical section and acts as follows:

- If it wants to enter a critical section, it keeps the token, enters the critical section, and exits from the critical section after finishing its work in the critical section. It then passes the token along the ring to its neighbor process. Note that the process can enter only one critical section when it receives the token. If it wants to enter another critical section, it must wait until it gets the token again.
- If it does not want to enter a critical section, it just passes the token along the ring to its neighbor process. Therefore, if none of the processes is interested in entering a critical section, the token simply keeps circulating around the ring.

Mutual exclusion is guaranteed by the algorithm because at any instance of time only one process can be in a critical section, since there is only a single token. Furthermore, since the ring is unidirectional and a process is permitted to enter only one critical section each time it gets the token, starvation cannot occur. In this algorithm the number of messages per critical section entry may vary from 1 (when every process always wants to enter a critical section) to an unbounded value (when no process wants to enter a critical section). Moreover, for a total of n processes in the system, the waiting time from the moment a process wants to enter a critical section until its actual entry may vary from the time needed to exchange 0 to $n-1$ token-passing messages. Zero token-passing messages are needed when the process receives the token just when it wants to enter the critical section, whereas $n-1$ messages are needed when the process wants to enter the critical section just after it has passed the token to its neighbor process.

The algorithm, however, requires the handling of the following types of failures:

1. *Process failure.* A process failure in the system causes the logical ring to break. In such a situation, a new logical ring must be established to ensure the continued circulation of the token among other processes. This requires detection of a failed process and dynamic reconfiguration of the logical ring when a failed process is detected or when a failed process recovers after failure.

Detection of a failed process can be easily done by making it a rule that a process receiving the token from its neighbor always sends an acknowledgment message to its neighbor. With this rule, a process detects that its neighbor has failed when it sends the token to it but does not receive the acknowledgment message within a fixed timeout period. On the other hand, dynamic reconfiguration of the logical ring can be done by maintaining the current ring configuration with each process. When a process detects that its neighbor has failed, it removes the failed process from the group by skipping it and passing the token to the process after it (actually to the next alive process in the sequence). When a process becomes alive after recovery, it simply informs the neighbor previous to it in the ring so that it gets the token during the next round of circulation.

2. *Lost token.* If the token is lost, a new token must be generated. Therefore, the algorithm must also have mechanisms to detect and regenerate a lost token. One method to solve this problem is to designate one of the processes on the ring as a "monitor" process. The monitor process periodically circulates a "who has the token?" message on the ring. This message rotates around the ring from one process to another. All processes simply pass this message to their neighbor process, except the process that has the token when it receives this message. This process writes its identifier in a special field of the message before passing it to its neighbor. When the message returns to the monitor process after one complete round, it checks the special field of the message. If there is no entry in this field, it concludes that the token has been lost, generates a new token, and circulates it around the ring.

There are two problems associated with this method—the monitor process may itself fail and the "who has the token?" message may itself get lost. Both problems may be solved by using more than one monitor processes. Each monitor process independently checks the availability of the token on the ring. However, when a monitor process

detects that the token is lost, it holds an election with other monitor processes to decide which monitor process will generate and circulate a new token (election algorithms are described later in this chapter). An election is needed to prevent the generation of multiple tokens that may happen when each monitor process independently detects that the token is lost, and each one generates a new token.

6.5 DEADLOCK

We saw in the previous section that there are several resources in a system for which the resource allocation policy must ensure exclusive access by a process. Since a system consists of a finite number of units of each resource type (for example, three printers, six tape drives, four disk drives, two CPUs, etc.), multiple concurrent processes normally have to compete to use a resource. In this situation, the sequence of events required to use a resource by a process is as follows:

1. *Request*. The process first makes a request for the resource. If the requested resource is not available, possibly because it is being used by another process, the requesting process must wait until the requested resource is allocated to it by the system. Note that if the system has multiple units of the requested resource type, the allocation of any unit of the type will satisfy the request. Also note that a process may request as many units of a resource as it requires with the restriction that the number of units requested may not exceed the total number of available units of the resource.

2. *Allocate*. The system allocates the resource to the requesting process as soon as possible. It maintains a table in which it records whether each resource is free or allocated and, if it is allocated, to which process. If the requested resource is currently allocated to another process, the requesting process is added to a queue of processes waiting for this resource. Once the system allocates the resource to the requesting process, that process can exclusively use the resource by operating on it.

3. *Release*. After the process has finished using the allocated resource, it releases the resource to the system. The system table records are updated at the time of allocation and release to reflect the current status of availability of resources.

The request and release of resources are system calls, such as *request* and *release* for devices, *open* and *close* for files, and *allocate* and *free* for memory space. Notice that of the three operations, *allocate* is the only operation that the system can control. The other two operations are initiated by a process.

With the above-mentioned pattern of request, allocation, and release of resources, if the total request made by multiple concurrent processes for resources of a certain type exceeds the amount available, some strategy is needed to order the assignment of resources in time. Care must be taken that the strategy applied cannot cause a deadlock, that is, a situation in which competing processes prevent their mutual progress even though no single one requests more resources than are available. It may

happen that some of the processes that entered the waiting state (because the requested resources were not available at the time of request) will never again change state, because the resources they have requested are held by other waiting processes. This situation is called *deadlock*, and the processes involved are said to be *deadlocked*. Hence, deadlock is the state of permanent blocking of a set of processes each of which is waiting for an event that only another process in the set can cause. All the processes in the set block permanently because all the processes are waiting and hence none of them will ever cause any of the events that could wake up any of the other members of the set.

A deadlock situation can be best explained with the help of an example. Suppose that a system has two tape drives T_1 and T_2 and the resource allocation strategy is such that a requested resource is immediately allocated to the requester if the resource is free. Also suppose that two concurrent processes P_1 and P_2 make requests for the tape drives in the following order:

1. P_1 requests for one tape drive and the system allocates T_1 to it.
2. P_2 requests for one tape drive and the system allocates T_2 to it.
3. P_1 requests for one more tape drive and enters a waiting state because no tape drive is presently available.
4. P_2 requests for one more tape drive and it also enters a waiting state because no tape drive is presently available.

From now on, P_1 and P_2 will wait for each other indefinitely, since P_1 will not release T_1 until it gets T_2 to carry out its designated task, that is, not until P_2 has released T_2, whereas P_2 will not release T_2 until it gets T_1. Therefore, the two processes are in a state of deadlock. Note that the requests made by the two processes are totally legal because each is requesting for only two tape drives, which is the total number of tape drives available in the system. However, the deadlock problem occurs because the total requests of both processes exceed the total number of units for the tape drive and the resource allocation policy is such that it immediately allocates a resource on request if the resource is free.

In the context of deadlocks, the term "resource" applies not only to physical objects (such as tape and disk drives, printers, CPU cycles, and memory space) but also to logical objects (such as a locked record in a database, files, tables, semaphores, and monitors). However, these resources should permit only exclusive use by a single process at a time and should be nonpreemptable. A *nonpreemptable resource* is one that cannot be taken away from a process to which it was allocated until the process voluntarily releases it. If taken away, it has ill effects on the computation already performed by the process. For example, a printer is a nonpreemptable resource because taking the printer away from a process that has started printing but has not yet completed its printing job and giving it to another process may produce printed output that contains a mixture of the output of the two processes. This is certainly unacceptable.

6.5.1 Necessary Conditions for Deadlock

Coffman et al. [1971] stated that the following conditions are necessary for a deadlock situation to occur in a system:

1. *Mutual-exclusion condition.* If a resource is held by a process, any other process requesting for that resource must wait until the resource has been released.
2. *Hold-and-wait condition.* Processes are allowed to request for new resources without releasing the resources that they are currently holding.
3. *No-preemption condition.* A resource that has been allocated to a process becomes available for allocation to another process only after it has been voluntarily released by the process holding it.
4. *Circular-wait condition.* Two or more processes must form a circular chain in which each process is waiting for a resource that is held by the next member of the chain.

All four conditions must hold simultaneously in a system for a deadlock to occur. If any one of them is absent, no deadlock can occur. Notice that the four conditions are not completely independent because the circular-wait condition implies the hold-and-wait condition. Although these four conditions are somewhat interrelated, it is quite useful to consider them separately to devise methods for deadlock prevention.

6.5.2 Deadlock Modeling

Deadlocks can be modeled using directed graphs. Before presenting a graphical model for deadlocks, some terminology from graph theory is needed:

1. *Directed graph.* A directed graph is a pair (N, E), where N is a nonempty set of nodes and E is a set of directed edges. A directed edge is an ordered pair (a, b), where a and b are nodes in N.
2. *Path.* A path is a sequence of nodes $(a, b, c, \ldots, i, j)$ of a directed graph such that $(a, b), (b, c), \ldots, (i, j)$ are directed edges. Obviously, a path contains at least two nodes.
3. *Cycle.* A cycle is a path whose first and last nodes are the same.
4. *Reachable set.* The reachable set of a node a is the set of all nodes b such that a path exists from a to b.
5. *Knot.* A knot is a nonempty set K of nodes such that the reachable set of each node in K is exactly the set K. A knot always contains one or more cycles.

An example of a directed graph is shown in Figure 6.8. The graph has a set of nodes $\{a, b, c, d, e, f\}$ and a set of directed edges $\{(a, b), (b, c), (c, d), (d, e), (e, f), (f, a), (e, b)\}$. It has two cycles (a, b, c, d, e, f, a) and (b, c, d, e, b). It also has a knot $\{a, b, c, d, e, f\}$ that contains the two cycles of the graph.

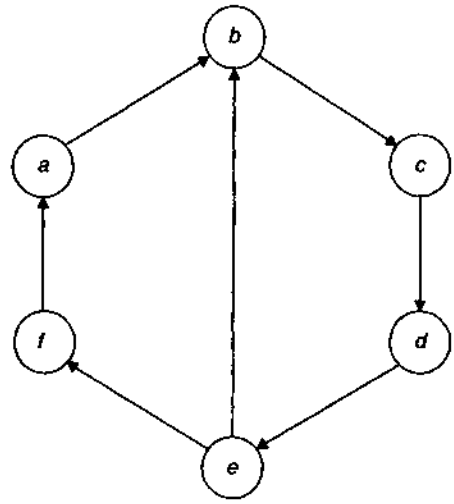

Fig. 6.8 A directed graph.

For deadlock modeling, a directed graph, called a *resource allocation graph*, is used in which both the set of nodes and the set of edges are partitioned into two types, resulting in the following graph elements:

1. *Process nodes.* A process node represents a process of the system. In a resource allocation graph, it is normally shown as a circle, with the name of the process written inside the circle (nodes P_1, P_2, and P_3 of Fig. 6.9).

2. *Resource nodes.* A resource node represents a resource of the system. In a resource allocation graph, it is normally shown as a rectangle with the name of the resource written inside the rectangle. Since a resource type R_j may have more than one unit in the system, each such unit is represented as a bullet within the rectangle. For instance, in the resource allocation graph of Figure 6.9, there are two units of resource R_1, one unit of R_2, and three units of R_3.

3. *Assignment edges.* An assignment edge is a directed edge from a resource node to a process node. It signifies that the resource is currently held by the process. In multiple units of a resource type, the tail of an assignment edge touches one of the bullets in the rectangle to indicate that only one unit of the resource is held by that process. Edges (R_1, P_1), (R_1, P_3), and (R_2, P_2) are the three assignment edges in the resource allocation graph of Figure 6.9.

4. *Request edges.* A request edge is a directed edge from a process node to a resource node. It signifies that the process made a request for a unit of the resource type and is currently waiting for that resource. Edges (P_1, R_2) and (P_2, R_1) are the two request edges in the resource allocation graph of Figure 6.9.

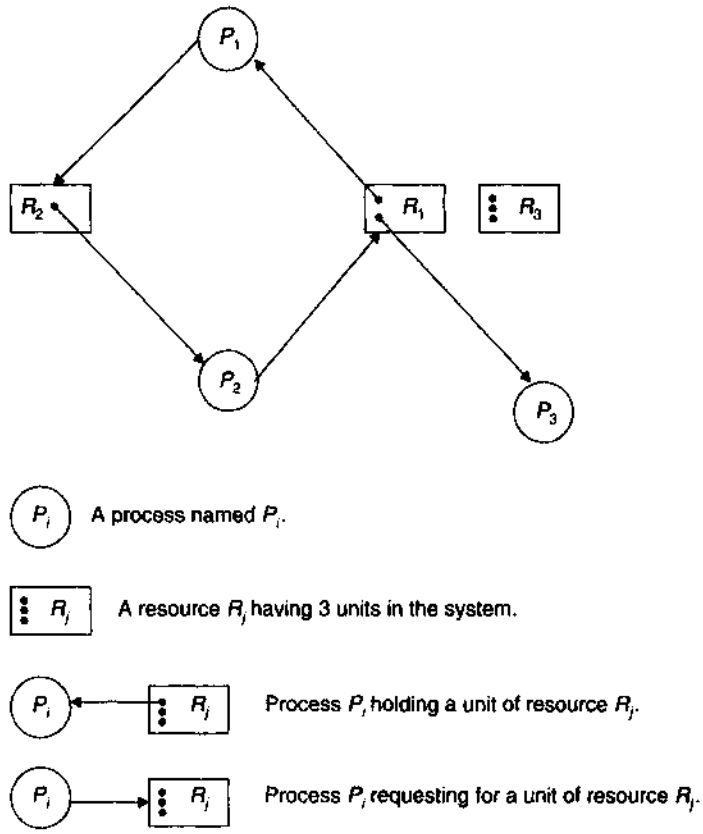

Fig. 6.9 Resource allocation graph.

Constructing a Resource Allocation Graph

A resource allocation graph provides an overall view of the processes holding or waiting for the various resources in the system. Therefore, the graph changes dynamically as the processes in the system request for or release resources or the system allocates a resource to a process. That is, when a process P_i requests for a unit of resource type R_j, a request edge (P_i, R_j) is inserted in the resource allocation graph. When this request can be fulfilled, a unit of resource R_j is allocated to P_i and the request edge (P_i, R_j) is instantaneously transformed to an assignment edge (R_j, P_i). Later, when P_i releases R_j, the assignment edge (R_j, P_i) is deleted from the graph.

Note that in many systems the resource allocation graph is not constructed in the above-mentioned manner. Rather it is used as a tool for making resource allocation decisions that do not lead to deadlock. In these systems, for the available resources, different request/release sequences are simulated step by step, and after every step, the graph is checked for deadlock. Later, in the resource allocation strategy, only those

sequences are allowed that do not lead to deadlock. Therefore, in these systems, the resource allocation graph is basically used to formulate a deadlock free resource allocation strategy.

Necessary and Sufficient Conditions for Deadlock

In a resource allocation graph, a cycle is a necessary condition for a deadlock to exist. That is, if the graph has no cycles, then it represents a state that is free from deadlock. On the other hand, if the graph contains a cycle, a deadlock may exist. Therefore, the presence of a cycle in a general resource allocation graph is a necessary but not a sufficient condition for the existence of deadlock. For instance, the resource allocation graph of Figure 6.9 contains a cycle (P_1, R_2, P_2, R_1, P_1) but does not represent a deadlock state. This is because when P_3 completes using R_1 and releases it, R_1 can be allocated to P_2. With both R_1 and R_2 allocated to it, P_2 can now complete its job after which it will release both R_1 and R_2. As soon as R_2 is released, it can be allocated to P_1. Therefore, all processes can finish their job one by one.

The sufficient condition for deadlock is different for the following different cases:

1. A cycle in the graph is both a necessary and a sufficient condition for deadlock if all the resource types requested by the processes forming the cycle have only a single unit each.

For example, the resource allocation graph of Figure 6.10 shows a deadlock state in which processes P_1 and P_2 are deadlocked. Notice that in this graph, although there are three units of resource R_3, it is not involved in the cycle (P_1, R_2, P_2, R_1, P_1). Both R_1 and R_2 that are involved in the cycle have only one unit each. Therefore, the cycle represents a deadlock state.

2. A cycle in the graph is a necessary but not a sufficient condition for deadlock if one or more of the resource types requested by the processes forming the cycle have more than one unit. In this case, a knot is a sufficient condition for deadlock.

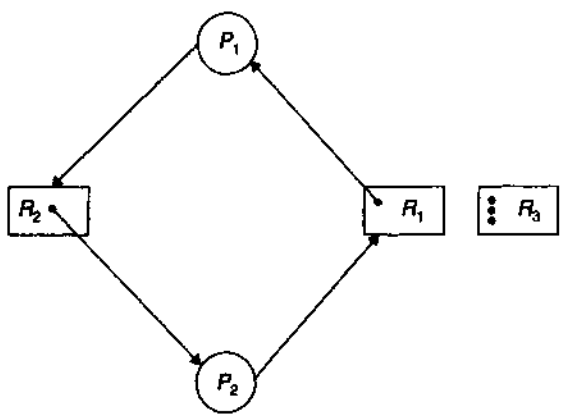

Fig. 6.10 A cycle representing a deadlock.

We have already seen that the cycle $(P_1, R_2, P_2, R_1, P_1)$ in the graph of Figure 6.9 does not represent a deadlock. This is because resource type R_1 has two units and there are no knots in the graph. Now suppose that in the same graph P_3 requests for R_2 and a request edge (P_3, R_2) is added to the graph. The modified graph is shown in Figure 6.11. This graph has two cycles $(P_1, R_2, P_2, R_1, P_1)$ and $(P_3, R_2, P_2, R_1, P_3)$ and a knot $\{P_1, P_2, P_3, R_1, R_2\}$. Since the graph contains a knot, it represents a deadlock state in which processes P_1, P_2, and P_3 are deadlocked.

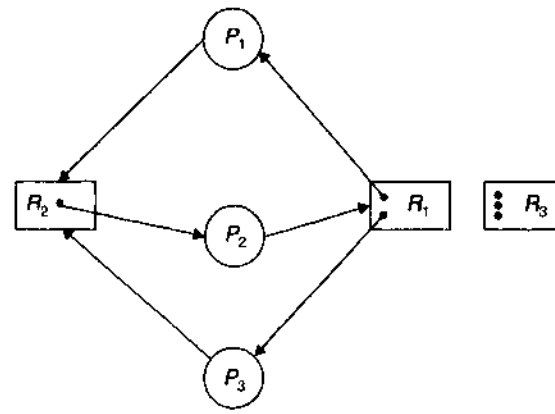

Fig. 6.11 A knot representing a deadlock.

In terms of the resource allocation graph, the necessary and sufficient conditions for deadlock can be summarized as follows:

- A cycle is a necessary condition for deadlock.
- If there is only a single unit of each resource type involved in the cycle, a cycle is both a necessary and a sufficient condition for a deadlock to exist.
- If one or more of the resource types involved in the cycle have more than one unit, a knot is a sufficient condition for a deadlock to exist.

Wait-for Graph

When all the resource types have only a single unit each, a simplified form of resource allocation graph is normally used. The simplified graph is obtained from the original resource allocation graph by removing the resource nodes and collapsing the appropriate edges. This simplification is based on the observation that a resource can always be identified by its current owner (process holding it). Figure 6.12 shows an example of a resource allocation graph and its simplified form.

The simplified graph is commonly known as a *wait-for graph* (*WFG*) because it clearly shows which processes are waiting for which other processes. For instance, in the WFG of Figure 6.12(*b*), processes P_1 and P_3 are waiting for P_2 and process P_2 is waiting for P_1. Since WFG is constructed only when each resource type has only a single unit, a cycle is both a necessary and sufficient condition for deadlock in a WFG.

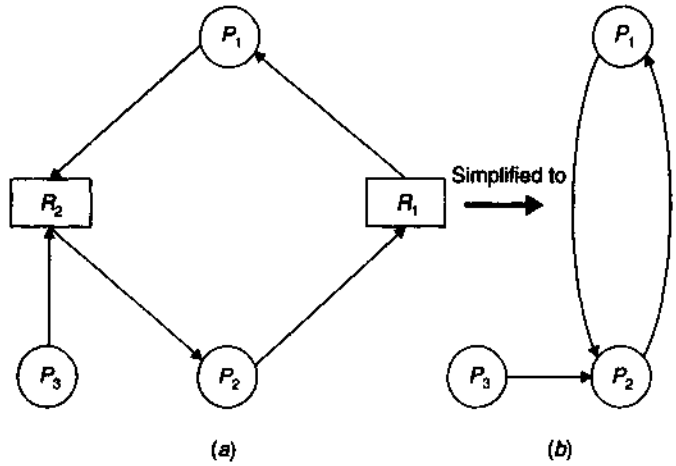

Fig. 6.12 A conversion from a resource allocation graph to a WFG: (*a*) resource allocation graph; (*b*) corresponding WFG.

6.5.3 Handling Deadlocks in Distributed Systems

In principle, deadlocks in distributed systems are similar to deadlocks in centralized systems. Therefore, the description of deadlocks presented above holds good both for centralized and distributed systems. However, handling of deadlocks in distributed systems is more complex than in centralized systems because the resources, the processes, and other relevant information are scattered on different nodes of the system.

Three commonly used strategies to handle deadlocks are as follows:

1. *Avoidance.* Resources are carefully allocated to avoid deadlocks.
2. *Prevention.* Constraints are imposed on the ways in which processes request resources in order to prevent deadlocks.
3. *Detection and recovery.* Deadlocks are allowed to occur and a detection algorithm is used to detect them. After a deadlock is detected, it is resolved by certain means.

Although the third strategy is the most commonly used one in distributed systems, for completion, the other two strategies will also be briefly described.

At this point, it may also be noted that some people prefer to make a distinction between two kinds of distributed deadlocks—*resource deadlocks* and *communication deadlocks*. As already described, a resource deadlock occurs when two or more processes wait permanently for resources held by each other. On the other hand, a communication deadlock occurs among a set of processes when they are blocked waiting for messages from other processes in the set in order to start execution but there are no messages in transit between them. When there are no messages in transit between any pair of processes in the set, none of the processes will ever receive a

message. This implies that all processes in the set are deadlocked. Communication deadlocks can be easily modeled by using WFGs to indicate which processes are waiting to receive messages from which other processes. Hence, the detection of communication deadlocks can be done in the same manner as that for systems having only one unit of each resource type.

Deadlock Avoidance

Deadlock avoidance methods use some advance knowledge of the resource usage of processes to predict the future state of the system for avoiding allocations that can eventually lead to a deadlock. Deadlock avoidance algorithms are usually in the following steps:

1. When a process requests for a resource, even if the resource is available for allocation, it is not immediately allocated to the process. Rather, the system simply assumes that the request is granted.
2. With the assumption made in step 1 and advance knowledge of the resource usage of processes, the system performs some analysis to decide whether granting the process's request is safe or unsafe.
3. The resource is allocated to the process only when the analysis of step 2 shows that it is safe to do so; otherwise the request is deferred.

Since the algorithms for deadlock avoidance are based on the concept of safe and unsafe states, it is important to look at the notion of safety in resource allocation. A system is said to be in a *safe state* if it is not in a deadlock state and there exists some ordering of the processes in which the resource requests of the processes can be granted to run all of them to completion. For a particular safe state there may be many such process orderings. Any ordering of the processes that can guarantee the completion of all the processes is called a *safe sequence*. The formation of a safe sequence is based on the idea of satisfying the condition that, for any process P_i in a safe sequence, the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all the processes lying before P_i in the safe sequence. This condition guarantees that process P_i can be run to completion because if the resources that P_i needs are not immediately available, P_i can wait until all other processes in the sequence lying before P_i have finished. When they have finished, P_i can obtain all its needed resources and run to completion. A system state is said to be *unsafe* if no safe sequence exists for that state.

The concept of safe and unsafe states can be best illustrated with the help of an example. Let us assume that in a system there are a total of 8 units of a particular resource type for which three processes P_1, P_2, and P_3 are competing. Suppose the maximum units of the resource required by P_1, P_2, and P_3 are 4, 5, and 6, respectively. Also suppose that currently each of the three processes is holding 2 units of the resource. Therefore, in the current state of the system, 2 units of the resource are free. The current state of the system can be modeled as shown in Figure 6.13(a).

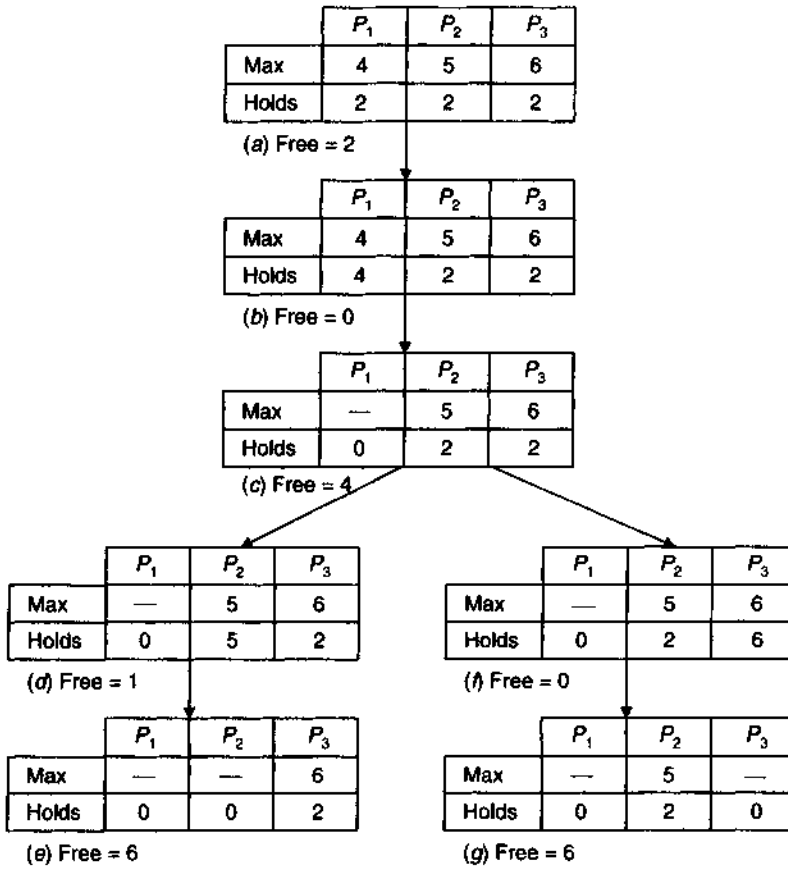

Fig. 6.13 Demonstration that the state in (*a*) is a safe state and has two safe sequences.

Now let us try to find out whether the state of Figure 6.13(*a*) is safe or unsafe. The analysis performed in Figure 6.13 shows that this state is safe because there exists a sequence of allocations that allows all processes to complete. In fact, as shown in the figure, for this state there are two safe sequences, (P_1, P_2, P_3) and (P_1, P_3, P_2). Let us see the scheduling of the resource units for the first of these two safe sequences. Starting from the state of Figure 6.13(*a*), the scheduler could simply run P_1 exclusively, until it asked for and got two more units of the resource that are currently free, leading to the state of Figure 6.13(*b*). When P_1 completes and releases the resources held by it, we get the state of Figure 6.13(*c*). Then the scheduler chooses to run P_2, eventually leading to the state of Figure 6.13(*d*). When P_2 completes and releases the resources held by it, the system enters the state of Figure 6.13(*e*). Now with the available resources, P_3 can be run to completion. The initial state of Figure 6.13(*a*) is a safe state because the system, by careful scheduling, can avoid deadlock. This example also shows that for a particular state there may be more than one safe sequence.

If resource allocation is not done cautiously, the system may move from a safe state to an unsafe state. For instance, let us consider the example shown in Figure 6.14. Figure 6.14(*a*) is the same initial state as that of Figure 6.13(*a*). This time, suppose process P_2 requests for one additional unit of the resource and the same is allocated to it by the system. The resulting system state is shown in Figure 6.14(*b*). The system is no longer in a safe state because we only have one unit of the resource free, which is not sufficient to run any of the three processes to completion. Therefore, there is no safe sequence for the state of Figure 6.14(*b*). Thus the decision to allocate one unit of the resource to P_2 moved the system from a safe state to an unsafe state.

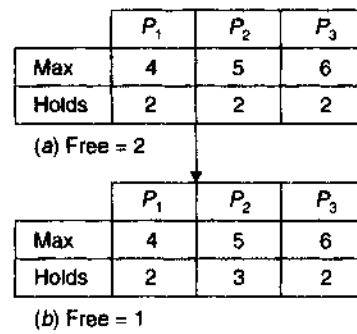

Fig. 6.14 Demonstration that an allocation may move the system from a safe to an unsafe state.

It is important to note the following remarks about safe and unsafe states:

1. The initial state in which no resources are yet allocated and all are available (free) is always a safe state.
2. From a safe state, the system can guarantee that all processes can be run to completion.
3. An unsafe state is not a deadlock state, but it may lead to a deadlock state. That is, from an unsafe state, the system cannot guarantee that all processes can be run to completion.

Deadlock avoidance algorithms basically perform resource allocation in such a manner as to ensure that the system will always remain in a safe state. Since the initial state of a system is always a safe state, whenever a process requests a resource that is currently available, the system checks to find out if the allocation of the resource to the process will change the state of the system from safe to unsafe. If no, the request is immediately granted; otherwise it is deferred.

Although theoretically attractive, deadlock avoidance algorithms are rarely used in practice due to the following reasons:

1. The algorithms work on the assumption that advance knowledge of the resource requirements of the various processes is available. However, in practice, processes rarely know in advance what their maximum resource needs will be. Modern operating systems

are attempting to provide more and more user-friendly interfaces, and as a result, it is becoming common to have users who do not have the slightest idea about what their resource needs are.

2. The algorithms also assume that the number of processes that compete for a particular resource is fixed and known in advance. However, in practice, the number of processes is not fixed but dynamically varies as new users log in and log out.

3. The algorithms also assume that the number of units of a particular resource type is always fixed and known in advance. However, in practice, the actual number of units available may change dynamically due to the sudden breakdown and repair of one or more units.

4. The manner in which these algorithms work restricts resource allocation too severely and consequently degrades the system performance considerably. This is because the algorithms first consider the worst possible case and then guarantee that the system is deadlock free even in the worst situation. This worst situation may arise but would be very unlikely. Thus many safe requests could be turned down.

The practical limitations of deadlock avoidance algorithms become more severe in a distributed system because the collection of information needed for making resource allocation decisions at one point is difficult and inefficient. Therefore, the deadlock avoidance strategy is never used in distributed operating systems.

Deadlock Prevention

This approach is based on the idea of designing the system in such a way that deadlocks become impossible. It differs from avoidance and detection in that no runtime testing of potential allocations need be performed.

We saw that mutual-exclusion, hold-and-wait, no-preemption, and circular-wait are the four necessary conditions for a deadlock to occur in a system. Therefore, if we can somehow ensure that at least one of these conditions is never satisfied, deadlocks will be impossible. Based on this idea, there are three important deadlock-prevention methods—collective requests, ordered requests, and preemption. The first one denies the hold-and-wait condition, the second one denies the circular-wait condition, and the third one denies the no-preemption condition.

The mutual-exclusion condition can also be denied for some nonsharable resources by devising an alternative way of using them. For instance, the following example, taken from [Tanenbaum 1992], illustrates how this can be done for a printer. A printer is a nonsharable resource. However, by spooling printer output, several processes can generate output at the same time. The spooled outputs are transferred one by one to the printer by the printer daemon. Therefore, the printer daemon is the only process that actually requests for the physical printer. Since the daemon never requests for any other resources, deadlock for the printer becomes structurally impossible. Unfortunately, in general, it is not possible to prevent deadlocks by denying the mutual-exclusion condition because some resources are intrinsically nonsharable and it is not possible to devise an alternative

way of using them. Therefore, denial of the mutual-exclusion condition for deadlock prevention is rarely used. The other three methods that are more commonly used are described below.

Collective Requests. This method denies the hold-and-wait condition by ensuring that whenever a process requests a resource, it does not hold any other resources. One of the following resource allocation policies may be used to ensure this:

1. A process must request all of its resources before it begins execution. If all the needed resources are available, they are allocated to the process so that the process can run to completion. If one or more of the requested resources are not available, none will be allocated and the process would just wait.

2. Instead of requesting all its resources before its execution starts, a process may request resources during its execution if it obeys the rule that it requests resources only when it holds no other resources. If the process is holding some resources, it can adhere to this rule by first releasing all of them and then re-requesting all the necessary resources.

The second policy has the following advantages over the first one:

1. In practice, many processes do not know how many resources they will need until they have started running. For such cases, the second approach is more useful.
2. A long process may require some resources only toward the end of its execution. In the first policy, the process will unnecessarily hold these resources for the entire duration of its execution. In the second policy, however, the process can request for these resources only when it needs them.

The collective requests method of deadlock prevention is simple and effective but has the following problems:

1. It generally has low resource utilization because a process may hold many resources but may not actually use several of them for fairly long periods.
2. It may cause starvation of a process that needs many resources, but whenever it makes a request for the needed resources, one or more of the resources is not available.
3. The method also raises an accounting question. When a process holds resources for extended periods during which they are not needed, it is not clear who should pay the charge for the idled resources.

Ordered Requests. In this method, each resource type is assigned a unique global number to impose a total ordering of all resource types. Now a resource allocation policy is used according to which a process can request a resource at any time, but the process should not request a resource with a number lower than the number of any of the resources that it is already holding. That is, if a process holds a resource type whose number is i, it may request a resource type having the number j only if $j > i$. If the process needs several

units of the same resource type, it must issue a single request for all the units. It has been proven that with this rule the resource allocation graph can never have cycles (denying the circular-wait condition), and hence deadlock is impossible.

Note that this algorithm does not require that a process must acquire all its resources in strictly increasing sequence. For instance, a process holding two resources having numbers 3 and 7 may release the resource having number 7 before requesting a resource having number 5. This is allowed because when the process requests for the resource having number 5, it is not holding any resource having number larger than 5.

The ordering of resources is decided according to the natural usage pattern of the resources. For example, since the tape drive is usually needed before the printer, it would be reasonable to assign a lower number to the tape drive than to the printer. However, the natural ordering is not always the same for all jobs. Therefore, a job that matches the decided ordering can be expected to use resources efficiently but others would waste resources. Another difficulty is that once the ordering is decided, it will stay for a long time because the ordering is coded into programs. Reordering will require reprogramming of several jobs. However, reordering may become inevitable when new resources are added to the system. Despite these difficulties, the method of ordered requests is one of the most efficient methods for handling deadlocks.

Preemption. A *preemptable* resource is one whose state can be easily saved and restored later. Such a resource can be temporarily taken away from the process to which it is currently allocated without causing any harm to the computation performed so far by the process. The CPU registers and main memory are examples of preemptable resources. If the resources are preemptable, deadlocks can be prevented by using either of the following resource allocation policies that deny the no-preemption condition:

1. When a process requests for a resource that is not currently available, all the resources held by the process are taken away (preempted) from it and the process is blocked. The process is unblocked when the resource requested by it and the resources preempted from it become available and can be allocated to it.

2. When a process requests a resource that is not currently available, the system checks if the requested resource is currently held by a process that is blocked, waiting for some other resource. If so, the requested resource is taken away (preempted) from the waiting process and given to the requesting process. Otherwise, the requesting process is blocked and waits for the requested resource to become available. Some of the resources that this process is already holding may be taken away (preempted) from it while it is blocked, waiting for the allocation of the requested resource. The process is unblocked when the resource requested by it and any other resource preempted from it become available and can be allocated to it.

In general, the applicability of this method for deadlock prevention is extremely limited because it works only for preemptable resources. However, the availability of atomic transactions and global timestamps makes this method an attractive approach for deadlock prevention in distributed and database transaction processing systems. The transaction mechanism allows a transaction (process) to be aborted (killed) without any ill

effect (transactions are described in Chapter 9). This makes it possible to preempt resources from processes holding them without any harm.

In the transaction-based deadlock prevention method, each transaction is assigned a unique priority number by the system, and when two or more transactions compete for the same resource, their priority numbers are used to break the tie. For example, Lamport's algorithm may be used to generate systemwide globally unique timestamps, and each transaction may be assigned a unique timestamp when it is created. A transaction's timestamp may serve as its priority number; a transaction having lower value of timestamp may have higher priority because it is older.

Rosenkrantz et al. [1978] proposed the following deadlock prevention schemes based on this idea:

1. *Wait-die scheme.* In this scheme, if a transaction T_i requests a resource that is currently held by another transaction T_j, T_i is blocked (waits) if its timestamp is lower than that of T_j; otherwise it is aborted (dies). For example, suppose that of the three transactions T_1, T_2, and T_3, T_1 is the oldest (has the lowest timestamp value) and T_3 is the youngest (has the highest timestamp value). Now if T_1 requests a resource that is currently held by T_2, T_1 will be blocked and will wait until the resource is voluntarily released by T_2. On the other hand, if T_3 requests a resource held by T_2, T_3 will be aborted.

2. *Wait-wound scheme.* In this scheme, if a transaction T_i requests a resource currently held by another transaction T_j, T_i is blocked (waits) if its timestamp is larger than that of T_j; otherwise T_j is aborted (wounded by T_i). Once again considering the same example of transactions T_1, T_2, and T_3, if T_1 requests a resource held by T_2, the resource will be preempted by aborting T_2 and will be given to T_1. On the other hand, if T_3 requests a resource held by T_2, T_3 will be blocked and will wait until the resource is voluntarily released by T_2.

Notice that both schemes favor older transactions, which is quite justified because, in general, an older transaction has run for a longer period and has used more system resources than a younger transaction. However, the manner in which the two schemes treat a younger transaction is worth noticing. In the wait-die scheme, a younger transaction is aborted when it requests for a resource held by an older transaction. The aborted transaction will be restarted after a predetermined time and will be aborted again if the older transaction is still holding the resource. This cycle may be repeated several times before the younger transaction actually gets the resource. This problem of the wait-die scheme can be solved by using an implementation mechanism that ensures that an aborted transaction is restarted only when its requested resource becomes available.

On the other hand, in the wait-wound scheme, when a younger transaction is aborted (wounded) by an older transaction, it will be restarted after a predetermined time, and this time it will be blocked (will wait) if its preempted resource is being held by the older transaction. Therefore, the implementation of the wait-wound scheme is simpler than the wait-die scheme. Furthermore, to avoid starvation, it is important that in the implementation of both schemes, a transaction should not be assigned a new timestamp when it is restarted after being aborted (a younger transaction will become older as time passes and will not be aborted again and again).

Deadlock Detection

In this approach for deadlock handling, the system does not make any attempt to prevent deadlocks and allows processes to request resources and to wait for each other in an uncontrolled manner. Rather, it uses an algorithm that keeps examining the state of the system to determine whether a deadlock has occurred. When a deadlock is detected, the system takes some action to recover from the deadlock. Some methods for deadlock detection in distributed systems are presented below, and some of the ways to recover from a deadlock situation are presented in the next section.

In principle, deadlock detection algorithms are the same in both centralized and distributed systems. It is based on maintenance of information on resource allocation to various processes in the form of a resource allocation graph and searching for a cycle/knot in the graph depending on whether the system has single/multiple units of each resource type. However, for simplicity, in the following description we consider only the case of a single unit of each resource type. Therefore, the deadlock detection algorithms get simplified to maintaining WFG and searching for cycles in the WFG.

The following steps may be followed to construct the WFG for a distributed system:

1. Construct a separate WFG for each site of the system in the following manner. Using the convention of Figure 6.9, construct a resource allocation graph for all the resources located on this site. That is, in the resource allocation graph of a site, a resource node exists for all the local resources and a process node exists for all processes that are either holding or waiting for a resource of this site immaterial of whether the process is local or nonlocal.
2. Convert the resource allocation graph constructed in step 1 to a corresponding WFG by removing the resource nodes and collapsing the appropriate edges. It may be noted that step 1 and this step are mentioned here only for clarity of presentation. The actual algorithm may be designed to directly construct a WFG.
3. Take the union of the WFGs of all sites and construct a single global WFG.

Let us illustrate the procedure with the help of the simple example shown in Figure 6.15. Suppose that the system is comprised of only two sites (S_1 and S_2) with S_1 having two resources R_1 and R_2 and S_2 having one resource R_3. Also suppose that there are three processes (P_1, P_2, P_3) that are competing for the three resources in the following manner:

- P_1 is holding R_1 and requesting for R_3
- P_2 is holding R_2 and requesting for R_1
- P_3 is holding R_3 and requesting for R_2

The corresponding resource allocation graphs for the two sites are shown in Figure 6.15(a). Notice that processes P_1 and P_3 appear in the graph of both the sites because they have requested for resources on both sites. On the other hand, process P_2 appears only in the graph of site S_1 because both resources requested by it are on S_1.

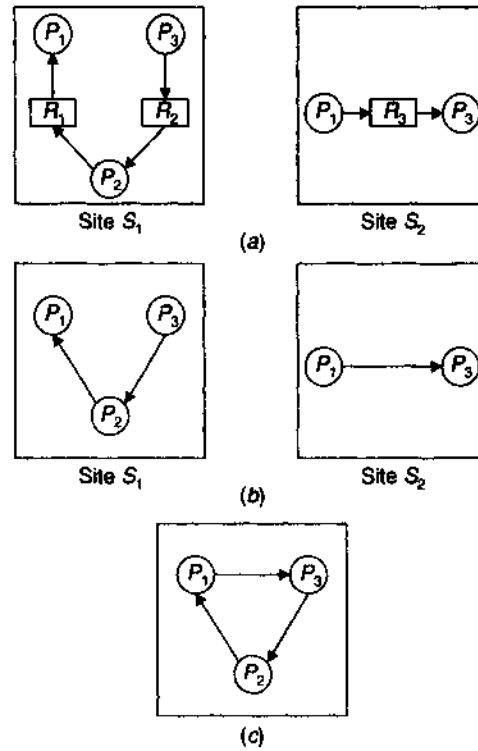

Fig. 6.15 Illustration of the construction of a WFG in a distributed system: (*a*) resource allocation graphs of each site; (*b*) WFGs corresponding to graphs in (*a*); (*c*) global WFG by taking the union of the two local WFGs of (*b*).

Figure 6.15(*b*) shows the corresponding WFGs for the two sites and Figure 6.15(*c*) shows the global WFG obtained by taking the union of the local WFGs of the two sites. Notice that although the local WFGs of the two sites do not contain any cycle, the global WFG contains a cycle, implying that the system is in a deadlock state. Therefore, this example shows that the local WFGs are not sufficient to characterize all deadlocks in a distributed system and the construction of a global WFG by taking the union of all local WFGs is required to finally conclude whether the system is in a state of deadlock or not.

The main difficulty in implementing deadlock detection in a distributed system is how to maintain the WFG. Three commonly used techniques for organizing the WFG in a distributed system are centralized, hierarchical, and distributed. These techniques are described below. However, before we describe them, it may be noted that one of the most important features of deadlock detection algorithms is correctness, which depends on the following properties [Knapp 1987]:

1. *Progress property.* This property states that all deadlocks must be detected in a finite amount of time.
2. *Safety property.* If a deadlock is detected, it must indeed exist. Message delays and out-of-date WFGs sometimes cause false cycles to be detected, resulting in the detection of deadlocks that do not actually exist. Such deadlocks are called *phantom deadlocks.*

Centralized Approach for Deadlock Detection. In the centralized deadlock detection approach, there is a local coordinator at each site that maintains a WFG for its local resources, and there is a central coordinator (also known as a centralized deadlock detector) that is responsible for constructing the union of all the individual WFGs. The central coordinator constructs the global WFG from information received from the local coordinators of all the sites. In this approach, deadlock detection is performed as follows:

1. If a cycle exists in the local WFG of any site, it represents a local deadlock. Such deadlocks are detected and resolved locally by the local coordinator of the site.
2. Deadlocks involving resources at two or more sites get reflected as cycles in the global WFG. Therefore, such deadlocks are detected and resolved by the central coordinator.

In the centralized approach, the local coordinators send local state information to the central coordinator in the form of messages. One of the following methods is used to transfer information from local coordinators to the central coordinator:

1. *Continuous transfer.* A local coordinator sends a message providing the update done in the local WFG whenever a new edge is added to or deleted from it.
2. *Periodic transfer.* To reduce the number of messages, a local coordinator periodically (when a number of changes have occurred in its local WFG) sends a list of edges added to or deleted from its WFG since the previous message was sent.
3. *Transfer-on-request.* A local coordinator sends a list of edges added to or deleted from its WFG since the previous message is sent only when the central coordinator makes a request for it. In this case, the central coordinator invokes the cycle detection algorithm periodically and requests information from each site just before invoking the algorithm.

Although the centralized deadlock detection approach is conceptually simple, it suffers from several drawbacks. First, it is vulnerable to failures of the central coordinator. Hence special provision for handling such faults have to be made. One approach is to provide a back-up central coordinator that duplicates the job of the central coordinator. Second, the centralized coordinator can constitute a performance bottleneck in large systems having too many sites. Third, the centralized coordinator may detect false deadlocks. Below we illustrate with a simple example how the algorithm may lead to the detection of false deadlocks and then we describe a method to overcome this third drawback.

Let us consider the same system configuration as that of Figure 6.15 and this time suppose that the three processes (P_1, P_2, P_3) compete for the three resources (R_1, R_2, R_3) in the following manner:

Step 1: P_1 requests for R_1 and R_1 is allocated to it.
Step 2: P_2 requests for R_2 and R_2 is allocated to it.
Step 3: P_3 requests for R_3 and R_3 is allocated to it.
Step 4: P_2 requests for R_1 and waits for it.

Step 5: P_3 requests for R_2 and waits for it.
Step 6: P_1 releases R_1 and R_1 is allocated to P_2.
Step 7: P_1 requests for R_3 and waits for it.

Assuming that the method of continuous transfer is employed by the algorithm, the following sequence of messages will be sent to the central coordinator:

m_1: from site S_1 to add the edge (R_1, P_1)
m_2: from site S_1 to add the edge (R_2, P_2)
m_3: from site S_2 to add the edge (R_3, P_3)
m_4: from site S_1 to add the edge (P_2, R_1)
m_5: from site S_1 to add the edge (P_3, R_2)
m_6: from site S_1 to delete edges (R_1, P_1) and (P_2, R_1), and add edge (R_1, P_2)
m_7: from site S_2 to add edge (P_1, R_3)

The resource allocation graphs maintained by the local coordinators of the two sites and the central coordinator are shown in Figure 6.16 (for clarity of presentation, the resource allocation graphs are shown instead of the WFGs). Figure 6.16(a) shows the graphs after step 5, that is, after message m_5 has been received by the central coordinator, and Figure 6.16(b) shows the graphs after message m_7 has been received by the central coordinator. The graph of the central coordinator in Figure 6.16(b) has no cycles, indicating that the system is free from deadlocks. However, suppose that message m_7 from site S_2 is received before message m_6 from site S_1 by the central coordinator. In this case, the central coordinator's view of the system will be as shown in the resource allocation graph of Figure 6.16(c). Therefore, the central coordinator will incorrectly conclude that a deadlock has occurred and may initiate deadlock recovery actions. Although the above example shows the possibility of detection of phantom deadlocks when the method of continuous transfer of information is used, phantom deadlocks may even get detected in the other two methods of information transfer due to incomplete or delayed information.

One method to avoid the detection of false deadlocks is to use Lamport's algorithm to append a unique global timestamp with each message. In our above example, since message m_7 from site S_2 to the central coordinator is caused by the request from site S_1 (see step 7), message m_7 will have a later timestamp than message m_6. Now if the central coordinator receives message m_7 before m_6 and detects a false deadlock, before taking any action to resolve the deadlock, it first confirms if the detected deadlock is a real one. For confirmation, it broadcasts a message asking all sites if any site has a message with timestamp earlier than T for updation of the global WFG. On receiving this message, if a site has a message with timestamp earlier than T, it immediately sends it to the central coordinator; otherwise it simply sends a negative reply. After receiving replies from all the sites, the central coordinator updates the global WFG (if there are any update messages), and if the cycle detected before still exists, it concludes that the deadlock is a real one and

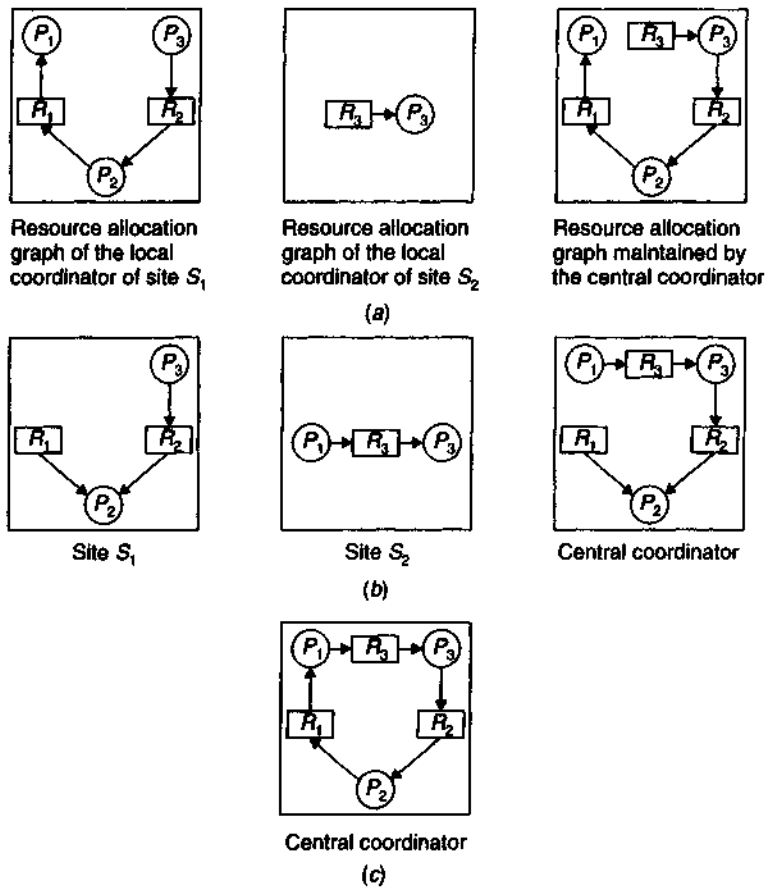

Fig. 6.16 Local and global resource allocation graphs in the centralized deadlock detection approach: (*a*) resource allocation graphs after step 5; (*b*) resource allocation graphs after step 7; (*c*) resource allocation graph of the central coordinator showing false deadlock if message m_7 is received before m_6 by the central coordinator.

initiates recovery actions. Notice that in our above example, in reply to its broadcast message, the central coordinator will receive message m_6 from site S_1 and a negative reply from site S_2. Therefore, after final updation of the global graph, the central coordinator's view of the system will change from that of Figure 6.16(*c*) to that in Figure 6.16(*b*). Hence no deadlock resolution action will be initiated.

Hierarchical Approach for Deadlock Detection. It has been observed that for typical applications most WFG cycles are very short. In particular, experimental measurements have shown that 90% of all deadlock cycles involve only two processes [Gray et al. 1981]. Therefore, the centralized approach seems to be less attractive for most real applications because of the significant time and message overhead involved in

assembling all the local WFGs at the central coordinator. Furthermore, to minimize communications cost, in geographically distributed systems, deadlock should be detected by a site located as close as possible to the sites involved in the cycle. But this is not possible in the centralized approach. The hierarchical approach overcomes these and other drawbacks of the centralized approach.

The hierarchical deadlock detection approach uses a logical hierarchy (tree) of deadlock detectors. These deadlock detectors are called *controllers*. Each controller is responsible for detecting only those deadlocks that involve the sites falling within its range in the hierarchy. Therefore, unlike the centralized approach in which the entire global WFG is maintained at a single site, in the hierarchical approach it is distributed over a number of different controllers. Each site has its own local controller that maintains its own local graph.

In the tree representing the hierarchy of controllers, the WFG to be maintained by a particular controller is decided according to the following rules:

1. Each controller that forms a leaf of the hierarchy tree maintains the local WFG of a single site.
2. Each nonleaf controller maintains a WFG that is the union of the WFGs of its immediate children in the hierarchy tree.

The lowest level controller that finds a cycle in its WFG detects a deadlock and takes necessary action to resolve it. Therefore, a WFG that contains a cycle will never be passed as it is to a higher level controller.

Let us illustrate the method with the help of the example shown in Figure 6.17. There are four sites and seven controllers in the system. Controllers A, B, C, and D maintain the local WFGs of sites S_1, S_2, S_3, and S_4, respectively. They form the leaves of the controllers' hierarchy tree. Controller E, being the parent of controllers A and B, maintains the union of the WFGs of controllers A and B. Similarly, controller F maintains the union of the WFGs of controllers C and D. Finally, controller G maintains the union of the WFGs of controllers E and F.

Notice from the figure that the deadlock cycle (P_1, P_3, P_2, P_1) that involves sites S_1 and S_2 gets reflected in the WFG of controller E, but the deadlock cycle (P_4, P_5, P_6, P_7, P_4) that involves sites S_2, S_3, and S_4 gets reflected only in the WFG of controller G. This is because controller G is the first controller in the hierarchy in whose range all the three sites S_2, S_3, and S_4 are covered. Also notice that although we have shown the deadlock cycle (P_1, P_3, P_2, P_1) in the WFG of controller G to reflect the union of the WFGs of controllers E and F, this will never happen in practice. This is because, when controller E detects the deadlock, it will initiate a recovery action instead of passing its WFG as it is to controller G.

Fully Distributed Approaches for Deadlock Detection. In the fully distributed deadlock detection approach, each site of the system shares equal responsibility for deadlock detection. Surveys of several algorithms based on this approach can be found in [Knapp 1987, Singhal 1989]. Below we describe two such algorithms. The first one is based on the construction of WFGs, and the second one is a probe-based algorithm.

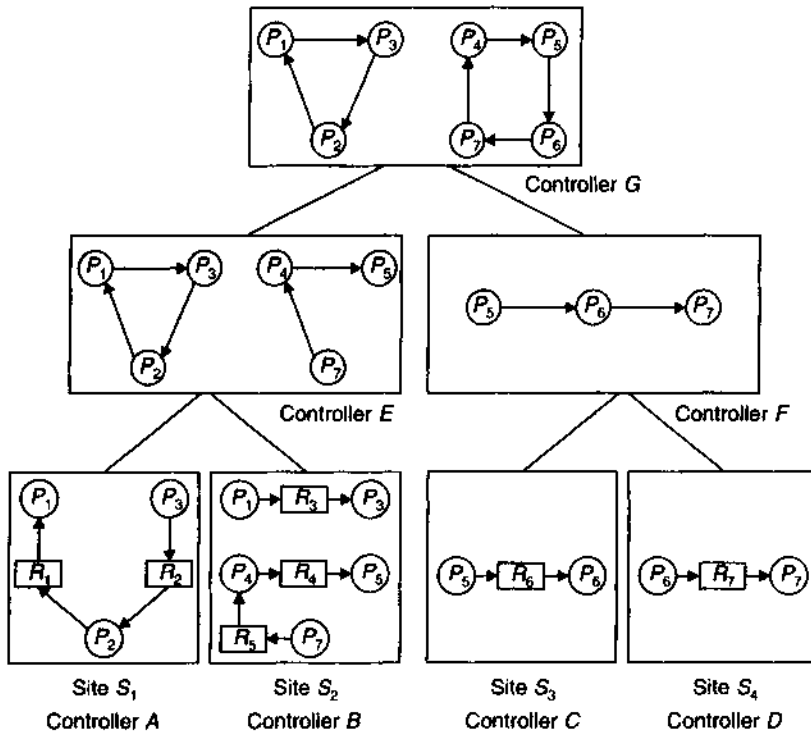

Fig. 6.17 Hierarchical deadlock detection approach.

WFG-Based Distributed Algorithm for Deadlock Detection. The description below follows from the description of the fully distributed deadlock detection algorithm presented in [Silberschatz and Galvin 1994]. As in the centralized and hierarchical approaches, in the WFG-based distributed algorithm, each site maintains its own local WFG. However, to model waiting situations that involve external (nonlocal) processes, a slightly modified form of WFG is used. In this modified WFG, an extra node P_{ex} is added to the local WFG of each site, and this node is connected to the WFG of the corresponding site in the following manner:

1. An edge (P_i, P_{ex}) is added if process P_i is waiting for a resource in another site being held by any process.
2. An edge (P_{ex}, P_j) is added if P_j is a process of another site that is waiting for a resource currently being held by a process of this site.

To illustrate the construction of this modified WFG, let us consider the example of Figure 6.18. In this example there are two sites, and the local WFGs of each site are shown in Figure 6.18(a). The modified WFGs of the two sites after the addition of node P_{ex} are shown in Figure 6.18(b). The explanation for the edges involving node P_{ex} are as follows:

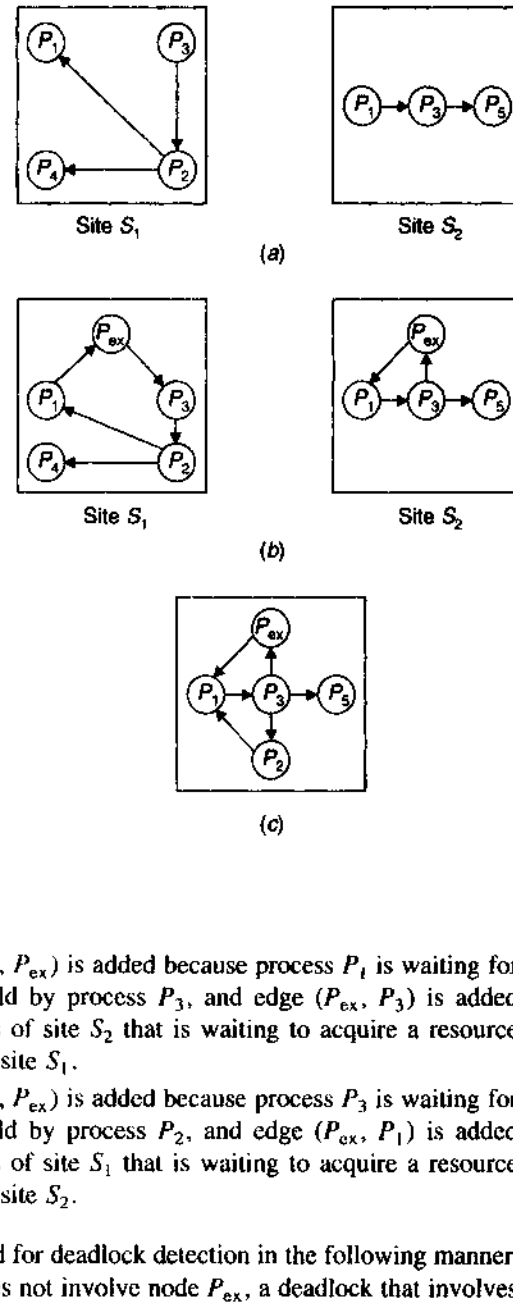

Fig. 6.18 Example illustrating the WFG-based fully distributed deadlock detection algorithm: (*a*) local WFGs; (*b*) local WFGs after addition of node P_{ex}; (*c*) updated local WFG of site S_2 after receiving the deadlock detection message from site S_1.

1. In the WFG of site S_1, edge (P_1, P_{ex}) is added because process P_1 is waiting for a resource in site S_2 that is held by process P_3, and edge (P_{ex}, P_3) is added because process P_3 is a process of site S_2 that is waiting to acquire a resource currently held by process P_2 of site S_1.
2. In the WFG of site S_2, edge (P_3, P_{ex}) is added because process P_3 is waiting for a resource in site S_1 that is held by process P_2, and edge (P_{ex}, P_1) is added because process P_1 is a process of site S_1 that is waiting to acquire a resource currently held by process P_3 of site S_2.

Now these modified WFGs are used for deadlock detection in the following manner. If a local WFG contains a cycle that does not involve node P_{ex}, a deadlock that involves only local processes of that site has occurred. Such deadlocks can be locally resolved without the need to consult any other site.

On the other hand, if a local WFG contains a cycle that involves node P_{ex}, there is a possibility of a distributed deadlock that involves processes of multiple sites. To confirm a distributed deadlock, a distributed deadlock detection algorithm is invoked by the site

whose WFG contains the cycle involving node P_{ex}. The algorithm works as described below.

Suppose a cycle involving node P_{ex} is detected in the WFG of site S_i. This cycle must be of the form

$$(P_{ex}, P_i, P_j, \ldots, P_k, P_{ex})$$

which means that process P_k is waiting for an external resource that belongs to some other site (say S_j). Therefore, site S_i sends a deadlock detection message to site S_j. This message does not contain the complete WFG of site S_i but only that part of the WFG that forms the cycle. For instance, in our example of Figure 6.18, if site S_1 detects its cycle first, it sends a message like $(P_{ex}, P_3, P_2, P_1, P_{ex})$ to site S_2 since P_1 is waiting for a resource in site S_2.

On receiving the message, site S_j updates its local WFG by adding those edges of the cycle that do not involve node P_{ex} to its WFG. That is, in our example, edges (P_3, P_2) and (P_2, P_1) will be added to the local WFG of site S_2, resulting in the new WFG of Figure 6.18(*c*).

Now if the newly constructed WFG of site S_j contains a cycle that does not involve node P_{ex}, a deadlock exists and an appropriate recovery procedure must be initiated. For instance, in our example, the newly constructed WFG of site S_2 contains a cycle (P_1, P_3, P_2, P_1) that does not involve node P_{ex}. Hence, in our example, the system is in a deadlock state.

On the other hand, if a cycle involving node P_{ex} is found in the newly constructed WFG of site S_j, S_j sends a deadlock detection message to the appropriate site (say S_k), and the whole procedure is repeated by site S_k. In this manner, after a finite number of deadlock detection message transfers from one site to another, either a deadlock is detected or the computation for deadlock detection halts.

A problem associated with the above algorithm is that two sites may initiate the deadlock detection algorithm independently for a deadlock that involves the same processes. For instance, in our example of Figure 6.18, sites S_1 and S_2 may almost simultaneously detect the cycles $(P_{ex}, P_3, P_2, P_1, P_{ex})$ and $(P_{ex}, P_1, P_3, P_{ex})$ respectively in their local WFGs, and both may send a deadlock detection message to the other site. The result will be that both sites will update their local WFGs and search for cycles. After detecting a deadlock, both may initiate a recovery procedure that may result in killing more processes than is actually required to resolve the deadlock. Furthermore, this problem also leads to extra overhead in unnecessary message transfers and duplication of deadlock detection jobs performed at the two sites.

One way to solve the above problem is to assign a unique identifier to each process P_i [denoted as $\mathrm{ID}(P_i)$]. Now when a cycle of the form $(P_{ex}, P_i, P_j, \ldots, P_k, P_{ex})$ is found in the local WFG of a site, this site initiates the deadlock detection algorithm by sending a deadlock detection message to the appropriate site only if

$$\mathrm{ID}(P_k) < \mathrm{ID}(P_i).$$

Otherwise, this site does not take any action and leaves the job of initiating the deadlock detection algorithm to some other site.

Let us apply the modified algorithm to our example of Figure 6.18. Let

$$ID(P_1) < ID(P_2) < ID(P_3) < ID(P_4) < ID(P_5)$$

Now suppose both sites S_1 and S_2 almost simultaneously detect the cycles (P_{ex}, P_3, P_2, P_1, P_{ex}) and (P_{ex}, P_1, P_3, P_{ex}), respectively, in their local WFGs. Since $ID(P_1) < ID(P_3)$, so site S_1 will initiate the deadlock detection algorithm. On the other hand, since $ID(P_3) > ID(P_1)$, site S_2 does not take any action on seeing the cycle in its local WFG. When site S_2 receives the deadlock detection message sent to it by site S_1, it updates its local WFG and searches for a cycle in the updated WFG. It detects the cycle (P_1, P_3, P_2, P_1) in the graph and then initiates a deadlock recovery procedure.

Probe-Based Distributed Algorithm for Deadlock Detection. The probe-based distributed deadlock detection algorithm described below was proposed by Chandy et al. [1983] and is known as the Chandy-Misra-Hass (or CMH) algorithm. It is considered to be the best algorithm to date for detecting global deadlocks in distributed systems. The algorithm allows a process to request for multiple resources at a time.

The algorithm is conceptually simple and works in the following manner. When a process that requests for a resource (or resources) fails to get the requested resource (or resources) and times out, it generates a special *probe* message and sends it to the process (or processes) holding the requested resource (or resources). The probe message contains the following fields (assuming that each process in the system is assigned a unique identifier):

1. The identifier of the process just blocked
2. The identifier of the process sending this message
3. The identifier of the process to whom this message is being sent

On receiving a probe message, the recipient checks to see if it itself is waiting for any resource (or resources). If not, this means that the recipient is using the resource requested by the process that sent the probe message to it. In this case, the recipient simply ignores the probe message. On the other hand, if the recipient is waiting for any resource (or resources), it passes the probe message to the process (or processes) holding the resource (or resources) for which it is waiting. However, before the probe message is forwarded, the recipient modifies its fields in the following manner:

1. The first field is left unchanged.
2. The recipient changes the second field to its own process identifier.
3. The third field is changed to the identifier of the process that will be the new recipient of this message.

Every new recipient of the probe message repeats this procedure. If the probe message returns back to the original sender (the process whose identifier is in the first field of the message), a cycle exists and the system is deadlocked.

Let us illustrate the algorithm with the help of the simple example shown in Figure 6.19. Notice that this figure depicts the same situation as that of Figure 6.18(*a*) but in a slightly different style. Suppose that process P_1 gets blocked when it requests for the resource held by process P_3. Therefore P_1 generates a probe message (P_1, P_1, P_3) and sends it to P_3. When P_3 receives this message, it discovers that it is itself blocked on processes P_2 and P_5. Therefore P_3 forwards the probes (P_1, P_3, P_2) and (P_1, P_3, P_5) to processes P_2 and P_5, respectively. When P_5 receives the probe message, it ignores it because it is not blocked on any other process. However, when P_2 receives the probe message, it discovers that it is itself blocked on processes P_1 and P_4. Therefore P_2 forwards the probes (P_1, P_2, P_1) and (P_1, P_2, P_4) to processes P_1 and P_4, respectively. Since the probe returns to its original sender (P_1), a cycle exists and the system is deadlocked.

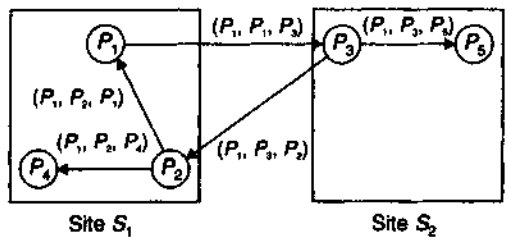

Fig. 6.19 Example illustrating the CMH distributed deadlock detection algorithm.

The CMH algorithm is popular, and variants of this algorithm are used in most distributed locking schemes due to the following attractive features of the algorithm:

1. The algorithm is easy to implement, since each message is of fixed length and requires few computational steps.
2. The overhead of the algorithm is fairly low.
3. There is no graph constructing and information collecting involved.
4. False deadlocks are not detected by the algorithm.
5. It does not require any particular structure among the processes.

Ways for Recovery from Deadlock

When a system chooses to use the detection and recovery strategy for handling deadlocks, it is not sufficient to simply detect deadlocks. The system must also have some way to recover from a detected deadlock. One of the following methods may be used in a system to recover from a deadlock:

- Asking for operator intervention
- Termination of process(es)
- Rollback of process(es)

Asking for Operator Intervention. The simplest way is to inform the operator that a deadlock has occurred and to let the operator deal with it manually. The system may assist the operator in decision making for recovery by providing him or her with a list of the processes involved in the deadlock.

This method is not suitable for use in modern systems because the concept of an operator continuously monitoring the smooth running of the system from the console has gradually vanished. Furthermore, although this method may work for a centralized system, it does not work in a distributed environment because when a deadlock involving processes of multiple sites is detected, it is not clear which site should be informed. If all the sites whose processes are involved in the deadlock are informed, each site's operator may independently take some action for recovery. On the other hand, if the operator of only a single site is informed, the operator may favor the process (or processes) of its own site while taking a recovery action. Furthermore, the operator of one site may not have the right to interfere with a process of another site for taking recovery action. Therefore, distributed systems normally use other methods described below in which the system recovers automatically from a deadlock.

Termination of Process(es). The simplest way to automatically recover from a deadlock is to terminate (kill) one or more processes and to reclaim the resources held by them, which can then be reallocated. Deadlock recovery algorithms based on this idea analyze the resource requirements and interdependencies of the processes involved in a deadlock cycle and then select a set of processes, which, if killed, can break the cycle.

Rollback of Process(es). Killing a process requires its restart from the very beginning, which proves to be very expensive, particularly when the process has already run for a substantially long time. To break a deadlock, it is sufficient to reclaim the needed resources from the processes that were selected for being killed. Also notice that to reclaim a resource from a process, it is sufficient to roll back the process to a point where the resource was not allocated to the process. The method of rollback is based on this idea.

In this method, processes are checkpointed periodically. That is, a process's state (its memory image and the list of resources held by it) is written to a file at regular intervals. Therefore, the file maintains a history of the process's states so that, if required, the process can be restarted from any of its checkpoints. Now when a deadlock is detected, the method described in the process termination approach is used to select a set of processes to be killed. However, this time, instead of total rollback (killing) of the selected processes, the processes are rolled back only as far as necessary to break the deadlock. That is, each selected process is rolled back to a checkpoint at which the needed resources can be reclaimed from it.

Although the rollback approach may appear to be less expensive than the process termination approach, this is not always true because of the extra overhead involved in the periodic checkpointing of all the processes. If deadlocks are rare in a system, it may be cheaper to use the process termination approach.

Issues in Recovery from Deadlock

Two important issues in the recovery action are selection of victims and use of transaction mechanism. These are described below.

Selection of Victim(s). In any of the recovery approaches described above, deadlock is broken by killing or rolling back one or more processes. These processes are called victims. Notice that even in the operator intervention approach, recovery involves killing one or more victims. Therefore, an important issue in any recovery procedure is to select the victims. Selection of victim(s) is normally based on two major factors:

1. *Minimization of recovery cost.* This factor suggests that those processes should be selected as victims whose termination/rollback will incur the minimum recovery cost. Unfortunately, it is not possible to have a universal cost function, and therefore, each system should determine its own cost function to select victims. Some of the factors that may be considered for this purpose are (a) the priority of the processes; (b) the nature of the processes, such as interactive or batch and possibility of rerun with no ill effects; (c) the number and types of resources held by the processes; (d) the length of service already received and the expected length of service further needed by the processes; and (e) the total number of processes that will be affected.

2. *Prevention of starvation.* If a system only aims at minimization of recovery cost, it may happen that the same process (probably because its priority is very low) is repeatedly selected as a victim and may never complete. This situation, known as *starvation*, must be somehow prevented in any practical system. One approach to handle this problem is to raise the priority of the process every time it is victimized. Another approach is to include the number of times a process is victimized as a parameter in the cost function.

Use of Transaction Mechanism. After a process is killed or rolled back for recovery from deadlock, it has to be rerun. However, rerunning a process may not always be safe, especially when the operations already performed by the process are nonidempotent. For example, if a process has updated the amount of a bank account by adding a certain amount to it, reexecution of the process will result in adding the same amount once again, leaving the balance in the account in an incorrect state. Therefore, the use of a transaction mechanism (which ensures all or no effect) becomes almost inevitable for most processes when the system chooses the method of detection and recovery for handling deadlocks. However, notice that the transaction mechanism need not be used for those processes that can be rerun with no ill effects. For example, rerun of a compilation process has no ill effects because all it does is read a source file and produce an object file.

6.6 ELECTION ALGORITHMS

Several distributed algorithms require that there be a *coordinator* process in the entire system that performs some type of coordination activity needed for the smooth running of other processes in the system. Two examples of such coordinator processes encountered

in this chapter are the coordinator in the centralized algorithm for mutual exclusion and the central coordinator in the centralized deadlock detection algorithm. Since all other processes in the system have to interact with the coordinator, they all must unanimously agree on who the coordinator is. Furthermore, if the coordinator process fails due to the failure of the site on which it is located, a new coordinator process must be elected to take up the job of the failed coordinator. *Election algorithms* are meant for electing a coordinator process from among the currently running processes in such a manner that at any instance of time there is a single coordinator for all processes in the system.

Election algorithms are based on the following assumptions:

1. Each process in the system has a unique priority number.
2. Whenever an election is held, the process having the highest priority number among the currently active processes is elected as the coordinator.
3. On recovery, a failed process can take appropriate actions to rejoin the set of active processes.

Therefore, whenever initiated, an election algorithm basically finds out which of the currently active processes has the highest priority number and then informs this to all other active processes. Different election algorithms differ in the way they do this. Two such election algorithms are described below. Readers interested in other election algorithms may refer to [Tel 1994]. For simplicity, in the description of both algorithms we will assume that there is only one process on each node of the distributed system.

6.6.1 The Bully Algorithm

This algorithm was proposed by Garcia-Molina [1982]. In this algorithm it is assumed that every process knows the priority number of every other process in the system. The algorithm works as follows.

When a process (say P_i) sends a *request* message to the coordinator and does not receive a reply within a fixed timeout period, it assumes that the coordinator has failed. It then initiates an election by sending an *election* message to every process with a higher priority number than itself. If P_i does not receive any response to its election message within a fixed timeout period, it assumes that among the currently active processes it has the highest priority number. Therefore it takes up the job of the coordinator and sends a message (let us call it a *coordinator* message) to all processes having lower priority numbers than itself, informing that from now on it is the new coordinator. On the other hand, if P_i receives a response for its election message, this means that some other process having higher priority number is alive. Therefore P_i does not take any further action and just waits to receive the final result (a coordinator message from the new coordinator) of the election it initiated.

When a process (say P_j) receives an election message (obviously from a process having a lower priority number than itself), it sends a response message (let us call it *alive* message) to the sender informing that it is alive and will take over the election activity. Now P_j holds an election if it is not already holding one. In this way, the election activity gradually moves on to the process that has the highest priority number among the

currently active processes and eventually wins the election and becomes the new coordinator.

As part of the recovery action, this method requires that a failed process (say P_k) must initiate an election on recovery. If the current coordinator's priority number is higher than that of P_k, then the current coordinator will win the election initiated by P_k and will continue to be the coordinator. On the other hand, if P_k's priority number is higher than that of the current coordinator, it will not receive any response for its election message. So it wins the election and takes over the coordinator's job from the currently active coordinator. Therefore, the active process having the highest priority number always wins the election. Hence the algorithm is called the "bully" algorithm. It may also be noted here that if the process having the highest priority number recovers after a failure, it does not initiate an election because it knows from its list of priority numbers that all other processes in the system have lower priority numbers than that of its own. Therefore, on recovery, it simply sends a coordinator message to all other processes and bullies the current coordinator into submission.

Let us now see the working of this algorithm with the help of an example. Suppose the system consists of five processes P_1, P_2, P_3, P_4, and P_5 and their priority numbers are 1, 2, 3, 4, and 5 respectively. Also suppose that at a particular instance of time the system is in a state in which P_2 is crashed, and P_1, P_3, P_4, and P_5 are active. Starting from this state, the functioning of the bully algorithm with the changing system states is illustrated below.

1. Obviously, P_5 is the coordinator in the starting state.
2. Suppose P_5 crashes.
3. Process P_3 sends a request message to P_5 and does not receive a reply within the fixed timeout period.
4. Process P_3 assumes that P_5 has crashed and initiates an election by sending an election message to P_4 and P_5 (recall that an election message is sent only to processes with higher priority numbers).
5. When P_4 receives P_3's election message, it sends an alive message to P_3, informing that it is alive and will take over the election activity. Process P_5 cannot respond to P_3's election message because it is down.
6. Now P_4 holds an election by sending an election message to P_5.
7. Process P_5 does not respond to P_4's election message because it is down, and therefore, P_4 wins the election and sends a coordinator message to P_1, P_2, and P_3, informing them that from now on it is the new coordinator. Obviously, this message is not received by P_2 because it is currently down.
8. Now suppose P_2 recovers from failure and initiates an election by sending an election message to P_3, P_4, and P_5. Since P_2's priority number is lower than that of P_4 (current coordinator), P_4 will win the election initiated by P_2 and will continue to be the coordinator.
9. Finally, suppose P_5 recovers from failure. Since P_5 is the process with the highest priority number, it simply sends a coordinator message to P_1, P_2, P_3, and P_4 and becomes the new coordinator.

6.6.2 A Ring Algorithm

The following algorithm is based on the ring-based election algorithms presented in [Tanenbaum 1995, Silberschatz and Galvin 1994]. In this algorithm it is assumed that all the processes in the system are organized in a logical ring. The ring is unidirectional in the sense that all messages related to the election algorithm are always passed only in one direction (clockwise/anticlockwise). Every process in the system knows the structure of the ring, so that while trying to circulate a message over the ring, if the successor of the sender process is down, the sender can skip over the successor, or the one after that, until an active member is located. The algorithm works as follows.

When a process (say P_i) sends a request message to the current coordinator and does not receive a reply within a fixed timeout period, it assumes that the coordinator has crashed. Therefore it initiates an election by sending an election message to its successor (actually to the first successor that is currently active). This message contains the priority number of process P_i. On receiving the election message, the successor appends its own priority number to the message and passes it on to the next active member in the ring. This member appends its own priority number to the message and forwards it to its own successor. In this manner, the election message circulates over the ring from one active process to another and eventually returns back to process P_i. Process P_i recognizes the message as its own election message by seeing that in the list of priority numbers held within the message the first priority number is its own priority number.

Note that when process P_i receives its own election message, the message contains the list of priority numbers of all processes that are currently active. Therefore of the processes in this list, it elects the process having the highest priority number as the new coordinator. It then circulates a coordinator message over the ring to inform all the other active processes who the new coordinator is. When the coordinator message comes back to process P_i after completing its one round along the ring, it is removed by process P_i. At this point all the active processes know who the current coordinator is.

When a process (say P_j) recovers after failure, it creates an *inquiry* message and sends it to its successor. The message contains the identity of process P_j. If the successor is not the current coordinator, it simply forwards the enquiry message to its own successor. In this way, the inquiry message moves forward along the ring until it reaches the current coordinator. On receiving an inquiry message, the current coordinator sends a reply to process P_j informing that it is the current coordinator.

Notice that in this algorithm two or more processes may almost simultaneously discover that the coordinator has crashed and then each one may circulate an election message over the ring. Although this results in a little waste of network bandwidth, it does not cause any problem because every process that initiated an election will receive the same list of active processes, and all of them will choose the same process as the new coordinator.

6.6.3 Discussion of the Two Election Algorithms

In the bully algorithm, when the process having the lowest priority number detects the coordinator's failure and initiates an election, in a system having total n processes,

altogether $n-2$ elections are performed one after another for the initiated one. That is, all the processes, except the active process with the highest priority number and the coordinator process that has just failed, perform elections by sending messages to all processes with higher priority numbers. Hence, in the worst case, the bully algorithm requires $O(n^2)$ messages. However, when the process having the priority number just below the failed coordinator detects that the coordinator has failed, it immediately elects itself as the coordinator and sends $n-2$ coordinator messages. Hence, in the best case, the bully algorithm requires only $n-2$ messages.

On the other hand, in the ring algorithm, irrespective of which process detects the failure of the coordinator and initiates an election, an election always requires $2(n-1)$ messages (assuming that only the coordinator process has failed); $n-1$ messages are needed for one round rotation of the election message, and another $n-1$ messages are needed for one round rotation of the coordinator message.

Next let us consider the complexity involved in the recovery of a process. In the bully algorithm, a failed process must initiate an election on recovery. Therefore, once again depending on the priority number of the process that initiates the recovery action, the bully algorithm requires $O(n^2)$ messages in the worst case, and $n-1$ messages in the best case. On the other hand, in the ring algorithm, a failed process does not initiate an election on recovery but simply searches for the current coordinator. Hence, the ring algorithm requires only $n/2$ messages on an average for recovery action.

In conclusion, as compared to the bully algorithm, the ring algorithm is more efficient and easier to implement.

6.7 SUMMARY

Sharing system resources among multiple concurrent processes may be cooperative or competitive in nature. Both cooperative and competitive sharing require adherence to certain rules of behavior that guarantee that correct interaction occurs. The rules for enforcing correct interaction are implemented in the form of synchronization mechanisms. In this chapter we saw the synchronization issues in distributed systems and mechanisms to handle these issues.

For correct functioning of several distributed applications, the clocks of different nodes of a distributed system must be mutually synchronized as well as with the external world (physical clock). Clock synchronization algorithms used in distributed systems are broadly classified into two types—centralized and distributed. In the centralized approach, there is a time server node and the goal of the algorithm is to keep the clocks of all other nodes synchronized with the clock time of the time server node. In the distributed approach, clock synchronization is done either by global averaging or localized averaging of the clocks of various nodes of the system.

Lamport observed that for most applications clock synchronization is not required, and it is sufficient to ensure that all events that occur in a distributed system can be totally ordered in a manner that is consistent with an observed behavior. Therefore, he defined the happened-before relation and introduced the concept of logical clocks for ordering of events based on the happened-before relation. The happened-before relation, however, is

only a partial ordering on the set of all events in the system. Therefore, for total ordering on the set of all system events, Lamport proposed the use of any arbitrary total ordering of the processes.

There are several resources in a system for which exclusive access by a process must be ensured. This exclusiveness of access is called mutual exclusion between processes, and the sections of a program that need exclusive access to shared resources are referred to as critical sections. The three basic approaches used by different algorithms for implementing mutual exclusion in distributed systems are centralized, distributed, and token passing. In the centralized approach, one of the processes in the system is elected as the coordinator, which coordinates the entry to the critical sections. In the distributed approach, all processes that want to enter the same critical section cooperate with each other before reaching a decision on which process will enter the critical section next. In the token-passing approach, mutual exclusion is achieved by using a single token that is circulated among the processes in the system.

Deadlock is the state of permanent blocking of a set of processes each of which is waiting for an event that only another process in the set can cause. In principle, deadlocks in distributed systems are similar to deadlocks in centralized systems. However, handling of deadlocks in distributed systems is more complex than in centralized systems because the resources, the processes, and other relevant information are scattered on different nodes of the system.

The three commonly used strategies to handle deadlocks are avoidance, prevention, and detection and recovery. Deadlock avoidance methods use some advance knowledge of the resource usage of processes to predict the future state of the system for avoiding allocations that can eventually lead to a deadlock. Deadlock prevention consists of carefully designing the system so that deadlocks become impossible. In the detection and recovery approach, deadlocks are allowed to occur, are detected by the system, and then are recovered. Of the three approaches, detection and recovery is the recommended approach for handling deadlocks in distributed systems. The three approaches used for deadlock detection in distributed systems are centralized, hierarchical, and fully distributed. For recovery from a detected deadlock, a system may use one of the following methods: asking for operator intervention, termination of process(es), or rollback of process(es).

Several distributed algorithms require that there be a coordinator process in the entire system. Election algorithms are meant for electing a coordinator process from among the currently running processes. Two election algorithms that were described in this chapter are the bully algorithm and the ring algorithm.

EXERCISES

6.1. Write pseudocode for an algorithm that decides whether a given set of clocks are synchronized or not. What input parameters are needed in your algorithm?

6.2. How do clock synchronization issues differ in centralized and distributed computing systems?