Karpagam Academy of Higher Education
KARPAGAM UNIVERSITY
(Established Under Section 3 of UGC Act 1956)
Eachanari Post, Coimbatore – 641 021. INDIA
Phone : 0422-2611146, 2611082 Fax No : 0422 -2611043

YEAR : 2016-2018 (LATERAL ENTRY)

15CAP505N DISTRIBUTED COMPUTING 4H - 4C

Instruction Hours / week: L: 4 T: 0 P: 0 Marks: Internal: 40 External: 60 Total: 100
End Semester Exam: 3Hours

Scope: This course helps to distinguish between local programming (on a single machine) and distributed programming using multiple components via a network. This course tends to cover what designers and programmers need to consider in developing applications in a distributed parallel computing environment.

Objective: To help students to

- Understand the architecture and topology of network
- Understand the design process of a distributed systems
- Examine distributed and parallel computing operating system
- Know the need and challenges of distributed database

UNIT – I

Introduction : Distributed Computing – Relation to multiprocessor and multi computer systems- message passing systems versus shared memory systems – primitives for distributed computing. Distributed Computations : distributed program – global state of a distributed system – models of process communications.

UNIT – II

Message ordering and Group Communication : message ordering paradigm – Asynchronous execution with synchronous communication – classification of application level multicast algorithm – distributed multicast algorithm at the network layer.

UNIT – III

System model for distributed computation – termination detection using distributed snapshots – termination detection in a faulty distributed system – Distributed mutual execution algorithm : Lamport's algorithm – Token Based algorithm – Raymond's tree-based algorithm.

UNIT – IV

DeadLock : system model – models of deadlocks – Knapp's classification of distributed deadlock detection algorithms – Chandy model – stable and unstable predicates – distributed algorithms for conjunctive predicates .

UNIT – V

Distributed shred memory : Abstraction – memory consistency – shared memory mutual execution – register hierarchy and wait free simulations- issues in failure recovery – check point based recovery – Authentication : protocols – password-based authentication- authentication protocol failures.

SUGGESTED READINGS

1. Ajay. D. Kshemkalyani and Mukesh Singhal. Distributed Computing: Principles, Algorithms and Systems.

2. Uyless D. Black, (2004), "Data Communication and Distributed Networks", 3rd Edition, Prentice hall of India, New Delhi.

3. Joel M Crichlow, (1998), "An Introduction to Distributed and Parallel Computing", 1st Edition, Prentice – Hall Publication, New Delhi.

WEB SITES

- wikipedia.org/wiki/Distributed_computing
- www.webopedia.com/TERM/D/distributed_computing.html
- www.tech-faq.com/distributed-computing.shtml

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University Established Under Section 3 of UGC Act 1956)

Coimbatore – 641 021.

LECTURE PLAN
DEPARTMENT OF CS, CA&IT

STAFF NAME: Dr. VIJAYALAKSHMI.P
SUBJECT NAME: DISTRIBUTED COMPUTING SUB.CODE:17CAP505A
SEMESTER: V CLASS: III MCA

S.No	Lecture Duration Period	Topics to be Covered	Support Material/Page Nos
		UNIT-I	
1.	1	Planning to computer program	W3, T1:1
2.	1	Concept of problem solving	W3, T1: 1
3.	1	Problem definition	W3, T1: 1
4.	1	Program Design	T1: 3-4
5.	1	Debugging	T1: 4, T1:219-228
6.	1	Types of errors in Programming	T1: 4-6
7.	1	Documentation	W4, T1: 241-247
8.	1	Recapitulation and Discussion of important questions	
	Total No of Hours Planned For Unit I=8		

		UNIT-II	
1.	1	Techniques of problem solving: Flow charting	W2
2.	1	Decision Table	W1
3.	1	Algorithms	W5, T1: 142-143
4.	1	Structured Programming concepts	W3, T1:6-8
5.	1	Programming methodologies: top down and bottom up programming	W4
6.	1	Recapitulation and Discussion of important questions	
	Total No of Hours Planned For Unit II = 6		
		UNIT-III	
1.	1	Overview of programming	T1: 1-3
2.	1	Overview of programming – History of Python	T2: 4-6, W1
3.	1	Structure of Python Program	T1: 8, T2:1-2, W1
4.	1	Elements of Python	W2
5.	1	Recapitulation and Discussion of important questions	
	Total No of Hours Planned For Unit III = 5		
		UNIT-IV	
1.	1	Introduction to python: Python Interpreter	W3,R1:1-5
2.	1	Using Python as Calculator	W3
3.	1	Python shell- Indentation	W1, R1: 5-8
4.	1	Atoms- Identifier and Keywords, literals, Strings	T1:11-14, W4, T1:71-78, T2:65-66,W1
5.	1	Operators- Arithmetic operators, Relational operator, logical or Boolean operator	T1: 16-17 T1:35-37
6.	1	Operators- Assignment operators, Ternary operator, Bitwise Operator, Increment or Decrement operator	T2:13-14 T2:3-4

7.	1	Recapitulation and Discussion of important questions	
	Total No of Hours Planned For Unit IV=7		
		UNIT-V	
1.	1	Creating Python programs: Input and output statements	W1,T2:22
2.	1	Control statement- Branching and Looping	T2:33-38
3.	1	Control statement – Conditional statements, Exit function	T2:33-38, W1, W2
4.	1	Difference between break, continue and Pass	T2:38-39,W1
5.	1	Defining Functions	T1:21-28, T2:46-52
6.	1	Default arguments	T1:28-30, T2:59-60, W2
7.	1	Recapitulation and Discussion of important questions	
8.	1	Discussion on previous ESE question papers	
9.	1	Discussion on previous ESE question papers	
10.	1	Discussion on previous ESE question papers	
	Total No of Hours Planned for Unit V= 10		
Total Planned Hours	**36**		

SUGGESTED READINGS

1. Allen Downey, Jeffrey Elkner, Chris Meyers, (2012). How to think like a computer scientist : learning with Python , Freely available online.

2. Budd,T.,(2011).Exploring Python, (1sted.) TMH

WEBSITES

1. http://docs.python.org/3/tutorial/index.html.

2. http://interactivepython.org/courselib/static/pythonds.

3. http://www.ibiblio.org/g2swap/byteofpython/read/.

1.1 INTRODUCTION

A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved. Distributed systems have been in existence since the start of the universe. From a school of fish to a flock of birds and entire ecosystems of microorganisms, there is communication among mobile intelligent agents in nature. With the widespread proliferation of the Internet and the emerging global village, the notion of distributed computing systems as a useful and widely deployed tool is becoming a reality.

For computing systems, a distributed system has been characterized in one of several ways:

• Using one when the crash of a computer you have never heard of prevents you from doing work.

• A collection of computers that do not share common memory or a common physical clock, that communicate by a messages passing over a communication network, and where each computer has its own memory and runs its own operating system. Typically the computers are semi-autonomous and are loosely coupled while they cooperate to address a problem collectively.

• A collection of independent computers that appears to the users of the system as a single coherent computer.

• A term that describes a wide range of computers, from weakly coupled systems such as wide-area networks, to strongly coupled systems such as local area networks, to very strongly coupled systems such as multiprocessor systems.

A distributed system can be characterized as a collection of mostly autonomous processors communicating over a communication network and having the following features:

• **No common physical clock** This is an important assumption because it introduces the element of "distribution" in the system and gives rise to the inherent asynchrony amongst the processors.

No shared memory This is a key feature that requires message-passing for communication. This feature implies the absence of the common physical clock. It may be noted that a distributed system may still provide the abstraction of a common address space via the distributed shared memory abstraction. Several aspects of shared memory multiprocessor systems have also been studied in the distributed computing literature.

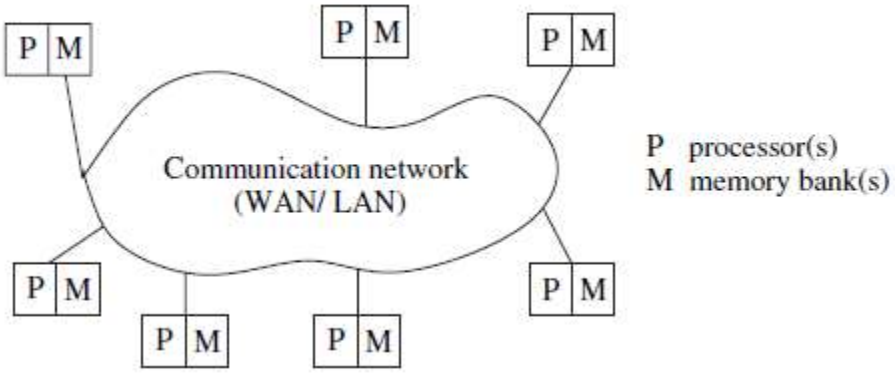
• **Geographical separation** The geographically wider apart that the processors are, the more representative is the system of a distributed system. However, it is not necessary for the processors to be on a wide-area network (WAN). Recently, the network/cluster of workstations (NOW/COW) configuration connecting processors on a LAN is also being increasingly regarded as a small distributed system. This NOW configuration is becoming popular because of the low-cost high-speed off-the-shelf processors now available. The Google search engine is based on the NOW architecture.

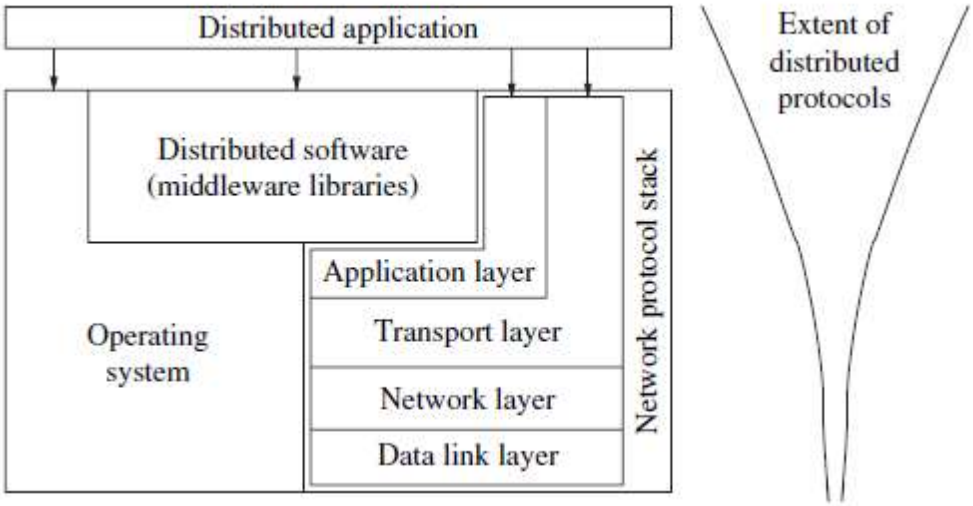
• **Autonomy and heterogeneity** The processors are "loosely coupled" in that they have different speeds and each can be running a different operating system. They are usually not part of a dedicated system, but cooperate with one another by offering services or solving a problem jointly.

Relation to computer system components

A typical distributed system is shown in Figure 1. Each computer has a memory-processing unit and the computers are connected by a communication network. Figure 2 shows the relationships of the software components that run on each of the computers and use the local operating system and network protocol stack for functioning. The distributed software is also termed as *middleware*. A *distributed execution* is the execution of processes across the distributed system to collaboratively achieve a common goal. An execution is also sometimes termed a *computation* or a *run*.

The distributed system uses a layered architecture to break down the complexity of system design. The middleware is the distributed software that





drives the distributed system, while providing transparency of heterogeneity at the platform level. Figure 2 schematically shows the interaction of this software with these system components at each processor. Here we assume that the middleware layer does not contain the traditional application layer functions of the network protocol stack, such as *http*, *mail*, *ftp*, and *telnet*. Various primitives and calls to functions defined in various libraries of the middleware layer are embedded in the user program code. There exist several libraries to choose from to invoke primitives for the more common functions – such as reliable and ordered multicasting – of the middleware layer.

There are several standards such as Object Management Group's (OMG) common object request broker architecture (CORBA), and the remote procedure call (RPC) mechanism. The RPC mechanism conceptually works like a local procedure call, with the difference that the procedure code may reside on a remote machine, and the RPC software sends a message across the network to invoke the remote procedure. It then awaits a reply, after which the procedure call completes from the perspective of the program that invoked it. Currently deployed commercial versions of middleware often use CORBA, DCOM (distributed component object model), Java, and RMI (remote method invocation) technologies. The message-passing interface (MPI) developed in the research community is an example of an interface for various communication functions.

The motivation for using a distributed system is some or all of the following requirements:

1. **Inherently distributed computations** In many applications such as money transfer in banking, or reaching consensus among parties that are geographically distant, the computation is inherently distributed.

2. **Resource sharing** Resources such as peripherals, complete data sets in databases, special libraries, as well as data (variable/files) cannot be fully replicated at all the sites because it is often neither practical nor cost-effective. Further, they cannot be placed at a single site because access to that site might prove to be a bottleneck. Therefore, such resources are typically distributed across the system. For example, distributed databases such as DB2 partition the data sets across several servers, in addition to replicating them at a few sites for rapid access as well as reliability.

3. **Access to geographically remote data and resources** In many scenarios, the data cannot be replicated at every site participating in the distributed execution because it may be too large or too sensitive to be replicated. For example, payroll data within a multinational corporation is both too large and too sensitive to be replicated at every branch office/site.

It is therefore stored at a central server which can be queried by branch offices. Similarly, special resources such as supercomputers exist only in certain locations, and to access such supercomputers, users need to log in remotely.

Advances in the design of resource-constrained mobile devices as well as in the wireless technology with which these devices communicate have given further impetus to the importance of distributed protocols and middleware.

4. **Enhanced reliability** A distributed system has the inherent potential to provide increased reliability because of the possibility of replicating resources and executions, as well as the reality that geographically distributed resources are not likely to crash/malfunction at the same time under normal circumstances. Reliability entails several aspects:

• availability, i.e., the resource should be accessible at all times;

• integrity, i.e., the value/state of the resource should be correct, in the face of concurrent access from multiple processors, as per the semantics expected by the application;

• fault-tolerance, i.e., the ability to recover from system failures, where such failures may be defined to occur in one of many failure models.

5. **Increased performance/cost ratio** By resource sharing and accessing geographically remote data and resources, the performance/cost ratio is increased. Although higher throughput has not necessarily been the main objective behind using a distributed system, nevertheless, any task can be partitioned across the various computers in the distributed

system. Such a configuration provides a better performance/cost ratio than using special parallel machines. This is particularly true of the NOW configuration. In addition to meeting the above requirements, a distributed system also offers the following advantages:

6. **Scalability** As the processors are usually connected by a wide-area network, adding more processors does not pose a direct bottleneck for the communication network.

7. **Modularity and incremental expandability** Heterogeneous processors may be easily added into the system without affecting the performance, as long as those processors are running the same middleware algorithms. Similarly, existing processors may be easily replaced by other processors.

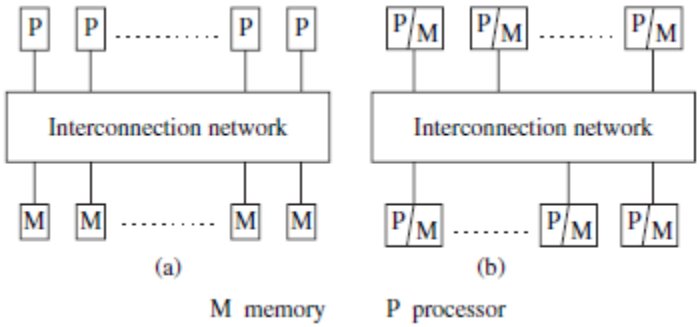
1.4 Relation to parallel multiprocessor/multicomputer systems

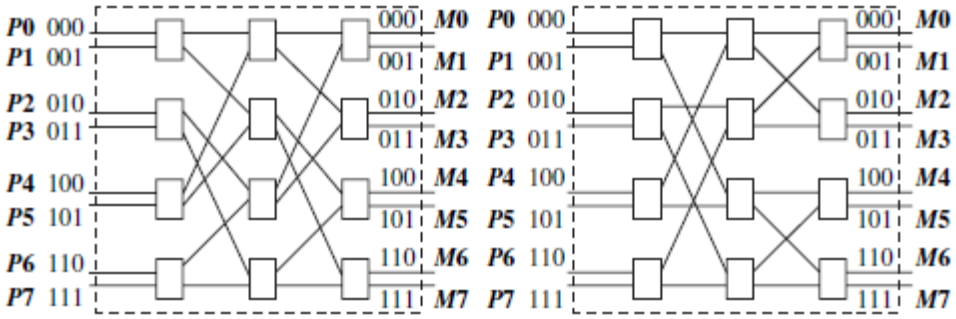
The characteristics of a distributed system were identified above. A typical distributed system would look as shown in Figure 1. However, how does one classify a system that meets some but not all of the characteristics? Is the system still a distributed system, or does it become a parallel multiprocessor system? To better answer these questions, we first examine the architecture of parallel systems, and then examine some well-known taxonomies for multiprocessor/multicomputer systems.

1.4.1 Characteristics of parallel systems

A parallel system may be broadly classified as belonging to one of three types:

1. A *multiprocessor system* is a parallel system in which the multiple processors have *direct access to shared memory* which forms a common address space. The architecture is shown in Figure 3(a). Such processors usually do not have a common clock. A multiprocessor system *usually* corresponds to a uniform memory access (UMA) architecture in which the access latency, i.e., waiting time, to complete an access to any memory location from any processor is the same. The processors are in very close physical proximity and are connected by an interconnection network. Interprocess communication across processors is traditionally through read and write operations on the shared memory, although the use of message-passing primitives such as those provided by





(a) 3-stage Omega network ($n = 8$, $M = 4$) (b) 3-stage Butterfly network ($n = 8$, $M = 4$)

e

the MPI, is also possible (using emulation on the shared memory). All the processors usually run the same operating system, and both the hardware and software are very tightly coupled. The processors are usually of the same type, and are housed within the same box/container with a shared memory. The interconnection network to access the memory may be a bus, although for greater efficiency, it is usually a *multistage switch* with a symmetric and regular design. Figure 4 shows two popular interconnection networks – the Omega network and the Butterfly network , each of which is a multi-stage network formed of 2×2 switching elements. Each 2×2 switch allows data on either of the two input wires to be switched to the upper or the lower output wire. In a single step, however, only one data unit can be sent on an output wire. So if the data from both the input wires is to be routed to the same output wire in a single step, there is a collision. Various techniques such as buffering or more elaborate interconnection designs can address collisions.

Each 2×2 switch is represented as a rectangle in the figure. Furthermore, a n-input and n-output network uses log n stages and log n bits for addressing. Routing in the 2×2 switch at stage k uses only the k^{th} bit, and hence can be done at clock speed in hardware. The multi-stage networks can be constructed recursively, and the interconnection pattern between any two stages can be expressed using an iterative or a recursive generating function. Besides the Omega and Butterfly (banyan) networks, other examples of multistage interconnection networks are the Clos and the shuffle-exchange networks. Each of these has very interesting mathematical properties that allow rich connectivity between the processor bank and memory bank.

1.4 Flynn's taxonomy

Flynn identified four processing modes, based on whether the processors execute the same or different instruction streams at the same time, and whether or not the processors processed the same (identical) data at the same time. It is instructive to examine this classification to understand the range of options used for configuring systems:

• Single instruction stream, single data stream (SISD)

This mode corresponds to the conventional processing in the von Neumann paradigm with a single CPU, and a single memory unit connected by a system bus.

• Single instruction stream, multiple data stream (SIMD)

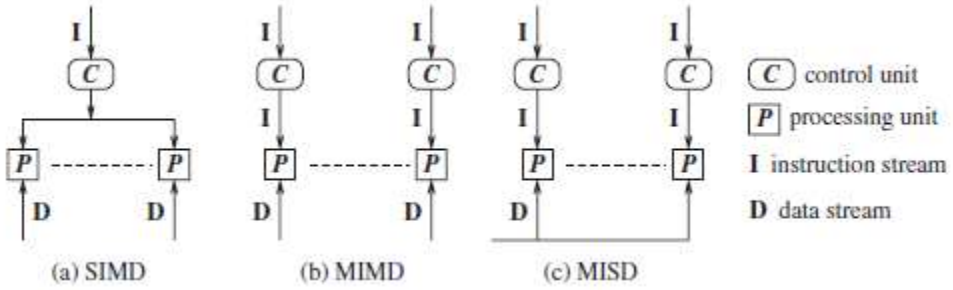
This mode corresponds to the processing by multiple homogenous processors which execute in lock-step on different data items. Applications that involve operations on large arrays and

matrices, such as scientific applications, can best exploit systems that provide the SIMD mode of operation because the data sets can be partitioned easily.

Several of the earliest parallel computers, such as Illiac-IV, MPP, CM2, and MasPar MP-1 were SIMD machines. Vector processors, array processors' and systolic arrays also belong to the SIMD class of processing. Recent SIMD architectures include co-processing units such as the MMX units in Intel processors (e.g., Pentium with the streaming SIMD extensions (SSE) options) and DSP chips such as the Sharc.

• Multiple instruction stream, single data stream (MISD)

This mode corresponds to the execution of different operations in parallel on the same data. This is a specialized mode of operation with limited but niche applications, e.g., visualization.



Multiple instruction stream, multiple data stream (MIMD)

In this mode, the various processors execute different code on different data. This is the mode of operation in distributed systems as well as in the vast majority of parallel systems. There is no common clock among the system processors. Sun Ultra servers, multicomputer PCs, and IBM SP machines are examples of machines that execute in MIMD mode.

SIMD, MISD, and MIMD architectures are illustrated in Figure 6. MIMD architectures are most general and allow much flexibility in partitioning code and data to be processed among the processors. MIMD architectures also include the classically understood mode of execution in distributed systems.

Coupling, parallelism, concurrency, and granularity

Coupling

The degree of coupling among a set of modules, whether hardware or software, is measured in terms of the interdependency and binding and/or homogeneity among the modules. When the degree of coupling is high (low), the modules are said to be tightly (loosely) coupled. SIMD and MISD architectures generally tend to be tightly coupled because of the common clocking of the shared instruction stream or the shared data stream. Here we briefly examine various MIMD architectures in terms of coupling:

• Tightly coupled multiprocessors (with UMA shared memory). These may be either switch-based (e.g., NYU Ultracomputer, RP3) or bus-based (e.g., Sequent, Encore).

• Tightly coupled multiprocessors (with NUMA shared memory or that communicate by message passing). Examples are the SGI Origin 2000 and the Sun Ultra HPC servers (that communicate via NUMA shared memory), and the hypercube and the torus (that communicate by message passing).

• Loosely coupled multicomputers (without shared memory) physically colocated. These may be bus-based (e.g., NOW connected by a LAN or Myrinet card) or using a more general

communication network, and the processors may be heterogeneous. In such systems, processors neither share memory nor have a common clock, and hence may be classified as distributed systems – however, the processors are very close to one another, which is characteristic of a parallel system. As the communication latency may be significantly lower than in wide-area distributed systems, the solution approaches to various problems may be different for such systems than for wide-area distributed systems.

• Loosely coupled multicomputers (without shared memory and without common clock) that are physically remote. These correspond to the conventional notion of distributed systems.

Parallelism or speedup of a program on a specific system

This is a measure of the relative speedup of a specific program, on a given machine. The speedup depends on the number of processors and the mapping of the code to the processors. It is expressed as the ratio of the time T_1_ with a single processor, to the time T_n_ with n processors.

Parallelism within a parallel/distributed program

This is an aggregate measure of the percentage of time that all the processors are executing CPU instructions productively, as opposed to waiting for communication (either via shared memory or message-passing) operations to complete. The term is traditionally used to characterize parallel programs. If the aggregate measure is a function of only the code, then the parallelism is independent of the architecture. Otherwise, this definition degenerates to the definition of parallelism in the previous section.

Concurrency of a program

This is a broader term that means roughly the same as parallelism of a program, but is used in the context of distributed programs. The *parallelism/concurrency* in a parallel/distributed program can be measured by the ratio of the number of local (non-communication and non-shared memory access) operations to the total number of operations, including the communication or shared memory access operations.

Granularity of a program

The ratio of the amount of computation to the amount of communication within the parallel/distributed program is termed as *granularity*. If the degree of parallelism is coarse-grained (fine-grained), there are relatively many more (fewer) productive CPU instruction executions, compared to the number of times the processors communicate either via shared memory or message passing and wait to get synchronized with the other processors. Programs with fine-grained parallelism are best suited for tightly coupled systems. These typically include SIMD and MISD architectures, tightly coupled MIMD multiprocessors (that have shared memory), and loosely coupled multicomputers (without shared memory) that are physically colocated. If programs with fine-grained parallelism were run over loosely coupled multiprocessors that are physically remote, the latency delays for the frequent communication over the WAN would significantly degrade the overall throughput. As a corollary, it follows that on such loosely coupled multicomputers, programs with a coarse-grained communication/message-passing granularity will incur substantially less overhead.

Figure 1.2 showed the relationships between the local operating system, the middleware implementing the distributed software, and the network protocol stack. Before moving on, we identify various classes of multiprocessor/ multicomputer operating systems:

• The operating system running on loosely coupled processors (i.e., heterogenous and/or geographically distant processors), which are themselves running loosely coupled software (i.e., software that is heterogenous), is classified as a *network operating system*. In this case, the application cannot run any significant distributed function that is not provided by the application layer of the network protocol stacks on the various processors.

• The operating system running on loosely coupled processors, which are running tightly coupled software (i.e., the middleware software on the processors is homogenous), is classified as a *distributed operating system*.

• The operating system running on tightly coupled processors, which are themselves running tightly coupled software, is classified as a *multiprocessor operating system*. Such a parallel system can run sophisticated algorithms contained in the tightly coupled software.

1.5 Message-passing systems versus shared memory systems

Shared memory systems are those in which there is a (common) shared address space throughout the system. Communication among processors takes place via shared data variables, and control variables for synchronization among the processors. Semaphores and monitors that were originally designed for shared memory uniprocessors and multiprocessors are examples of how synchronization can be achieved in shared memory systems. All multicomputer (NUMA as well as message-passing) systems that do not have a shared address space provided by the underlying architecture and hardware necessarily communicate by message passing. Conceptually, programmers find it easier to program using shared memory than by message passing. For this and several other reasons that we examine later, the abstraction called *shared memory* is sometimes provided to simulate a shared address space. For a distributed system, this abstraction is called *distributed shared memory*. Implementing this abstraction has a certain cost but it simplifies the task of the application programmer. There also exists a well-known folklore result that communication via message-passing can be simulated by communication via shared memory and vice-versa. Therefore, the two paradigms are equivalent.

1.5.1 Emulating message-passing on a shared memory system (MP →SM)

The shared address space can be partitioned into disjoint parts, one part being assigned to each processor. “Send” and “receive” operations can be implemented by writing to and reading from the destination/sender processor’s address space, respectively. Specifically, a separate location can be reserved as the mailbox for each ordered pair of processes. A Pi–Pj message-passing can be emulated by a write by Pi to the mailbox and then a read by Pj from the mailbox. In the simplest case, these mailboxes can be assumed to have unbounded size. The write and read operations need to be controlled using synchronization primitives to inform the receiver/sender after the data has been sent/received.

1.5.2 Emulating shared memory on a message-passing system (SM →MP)

This involves the use of "send" and "receive" operations for "write" and "read" operations. Each shared location can be modeled as a separate process; "write" to a shared location is emulated by sending an update message to the corresponding owner process; a "read" to a shared location is emulated by sending a query message to the owner process. As accessing another processor's memory requires send and receive operations, this emulation is expensive. Although emulating shared memory might seem to be more attractive from a programmer's perspective, it must be remembered that in a distributed system, it is only an abstraction. Thus, the latencies involved in read and write operations may be high even when using shared memory emulation because the read and write operations are implemented by using network-wide communication under the covers.

An application can of course use a combination of shared memory and message-passing. In a MIMD message-passing multicomputer system, each "processor" may be a tightly coupled multiprocessor system with shared memory. Within the multiprocessor system, the processors communicate via shared memory. Between two computers, the communication is by message passing. As message-passing systems are more common and more suited for wide-area distributed systems, we will consider message-passing systems more extensively than we consider shared memory systems.

1.6 Primitives for distributed communication

1.6.1 Blocking/non-blocking, synchronous/asynchronous primitives

Message send and message receive communication primitives are denoted *Send()* and *Receive()*, respectively. A *Send* primitive has at least two parameters – the destination, and the buffer in the user space, containing the data to be sent. Similarly, a *Receive* primitive has at least two parameters – the source from which the data is to be received (this could be a wildcard), and the user buffer into which the data is to be received.

There are two ways of sending data when the *Send* primitive is invoked – the buffered option and the unbuffered option. The *buffered option* which is the standard option copies the data from the user buffer to the kernel buffer. The data later gets copied from the kernel buffer onto the network. In the *unbuffered option*, the data gets copied directly from the user buffer onto the network. For the *Receive* primitive, the buffered option is usually required because the data may already have arrived when the primitive is invoked, and needs a storage place in the kernel.

The following are some definitions of blocking/non-blocking and synchronous/ asynchronous primitives:

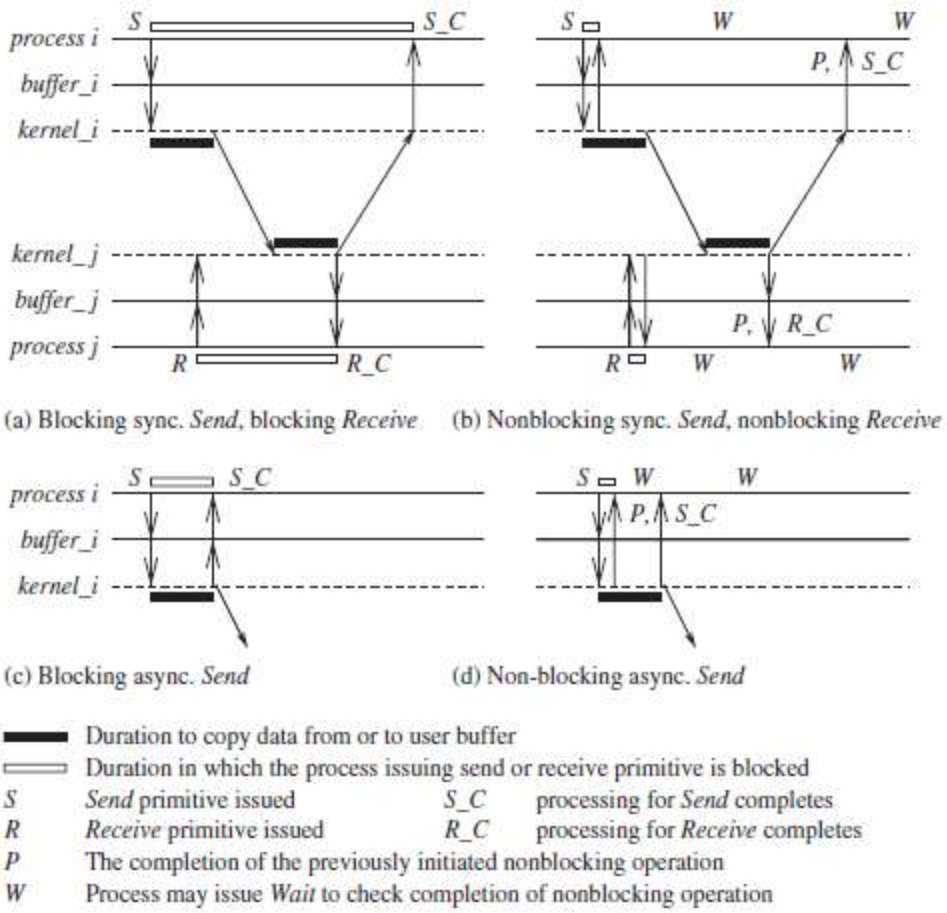
• **Synchronous primitives** A *Send* or a *Receive* primitive is *synchronous* if both the *Send()* and *Receive()* handshake with each other. The processing for the *Send* primitive completes only after the invoking processor learns that the other corresponding *Receive* primitive has also been invoked and that the receive operation has been completed. The processing for the *Receive* primitive completes when the data to be received is copied into the receiver's user buffer.

• **Asynchronous primitives** A *Send* primitive is said to be *asynchronous* if control returns back to the invoking process after the data item to be sent has been copied out of the user-specified buffer. It does not make sense to define asynchronous *Receive* primitives.

• **Blocking primitives** A primitive is *blocking* if control returns to the invoking process after the processing for the primitive (whether in synchronous or asynchronous mode) completes.
• **Non-blocking primitives** A primitive is *non-blocking* if control returns back to the invoking process immediately after invocation, even though the operation has not completed. For a non-blocking *Send*, control returns to the process even before the data is copied out of the user buffer. For a non-blocking *Receive*, control returns to the process even before the data may have arrived from the sender.
For non-blocking primitives, a return parameter on the primitive call returns a system-generated *handle* which can be later used to check the status of completion of the call. The process can check for the completion of the call in two ways. First, it can keep checking (in a loop or periodically)
if the handle has been flagged or *posted*. Second, it can issue a *Wait* with a list of handles as parameters. The *Wait* call usually blocks until one of the parameter handles is posted. Presumably after issuing the primitive in non-blocking mode, the process has done whatever actions it could and now needs to know the status of completion of the call, therefore using a blocking *Wait()* call is usual programming practice. The code for a non-blocking *Send* would look as shown in Figure 7.

```
Send(X, destination, handle_k)                    // handle_k is a return parameter
...
...
Wait(handle_1, handle_2, ..., handle_k, ..., handle_m)    // Wait always blocks
```

If at the time that *Wait()* is issued, the processing for the primitive (whether synchronous or asynchronous) has completed, the *Wait* returns immediately. The completion of the processing of the primitive is detectable by checking the value of handlek. If the processing of the primitive has not completed, the *Wait* blocks and waits for a signal to wake it up. When the processing for the primitive completes, the communication subsystem software sets the value of handlek and wakes up (signals) any process with a *Wait* call blocked on this handlek. This is called *posting* the completion of the operation.
There are therefore four versions of the *Send* primitive – synchronous blocking, synchronous non-blocking, asynchronous blocking, and asynchronous non-blocking. For the *Receive* primitive, there are the blocking synchronous and non-blocking synchronous versions. These versions of the primitives are illustrated in Figure 8 using a timing diagram. Here, three time lines are



check for the completion of the non-blocking *Receive* by invoking the *Wait* operation on the returned handle. (If the data has already arrived when the call is made, it would be pending in some kernel buffer, and still needs to be copied to the user buffer.)

A synchronous *Send* is easier to use from a programmer's perspective because the handshake between the *Send* and the *Receive* makes the communication appear instantaneous, thereby simplifying the program logic. The "instantaneity" is, of course, only an illusion, as can be seen from Figure 8(a) and (b). In fact, the *Receive* may not get issued until much after the data arrives at Pj , in which case the data arrived would have to be buffered in the system buffer at Pj and not in the user buffer. At the same time, the sender would remain blocked. Thus, a synchronous *Send* lowers the efficiency within process Pi.

The non-blocking asynchronous *Send* (see Figure 8(d)) is useful when a large data item is being sent because it allows the process to perform other instructions in parallel with the completion of the *Send*. The non-blocking synchronous *Send* (see Figure 8(b)) also avoids the potentially large delays for handshaking, particularly when the receiver has not yet issued the *Receive* call. The non-blocking *Receive* (see Figure 8(b)) is useful when a large data item is being received and/or when the sender has not yet issued the *Send* call, because it allows the process to perform other instructions in parallel with the completion of the *Receive*. Note that if the data has already arrived, it is stored in the kernel buffer, and it may take a while to copy

it to the user buffer specified in the *Receive* call. For non-blocking calls, however, the burden on the programmer increases because he or she has to keep track of the completion of such operations in order to meaningfully reuse (write to or read from) the user buffers. Thus, conceptually, blocking primitives are easier to use.

1.6. Processor synchrony

As opposed to the classification of synchronous and asynchronous communication primitives, there is also the classification of synchronous versus asynchronous processors. *Processor synchrony* indicates that all the processors execute in lock-step with their clocks synchronized. As this synchrony is not attainable in a distributed system, what is more generally indicated is that for a large granularity of code, usually termed as a *step*, the processors are synchronized. This abstraction is implemented using some form of barrier synchronization to ensure that no processor begins executing the next step of code until all the processors have completed executing the previous steps of code assigned to each of the processors.

A model of distributed computations

A distributed system consists of a set of processors that are connected by a communication network. The communication network provides the facility of information exchange among processors. The communication delay is finite but unpredictable. The processors do not share a common global memory and communicate solely by passing messages over the communication network. There is no physical global clock in the system to which processes have instantaneous access. The communication medium may deliver messages out of order, messages may be lost, garbled, or duplicated due to timeout and retransmission, processors may fail, and communication links may go down. The system can be modeled as a directed graph in which vertices represent the processes and edges represent unidirectional communication channels.

A distributed application runs as a collection of processes on a distributed system. This chapter presents a model of a distributed computation and introduces several terms, concepts, and notations that will be used in the subsequent chapters.

2.1 A distributed program

A distributed program is composed of a set of n asynchronous processes p1, p2, , pi, , pn that communicate by message passing over the communication network. Without loss of generality, we assume that each process is running on a different processor. The processes do not share a global memory and communicate solely by passing messages. Let Cij denote the channel from process pi to process pj and let mij denote a message sent by pi to pj . The communication delay is finite and unpredictable. Also, these processes do not share a global clock that is instantaneously accessible to these processes. Process execution and message transfer are asynchronous – a process may execute an action spontaneously and a process sending a message does not wait for the delivery of the message to be complete.

The global state of a distributed computation is composed of the states of the processes and the communication channels. The state of a process is characterized by the state of its local

memory and depends upon the context. The state of a channel is characterized by the set of messages in transit in the channel.

2.2 A model of distributed executions

The execution of a process consists of a sequential execution of its actions. The actions are atomic and the actions of a process are modeled as three ypes of events, namely, internal events, message send events, and message receive events. Let ex i denote the xth event at process pi. Subscripts and/or superscripts will be dropped when they are irrelevant or are clear from the context. For a message m, let send_m_ and rec_m_ denote its send and receive events, respectively. The occurrence of events changes the states of respective processes and channels, thus causing transitions in the global system state. An internal event changes the state of the process at which it occurs. A send event (or a receive event) changes the state of the process that sends (or receives) the message and the state of the channel on which the message is sent (or received). An internal event only affects the process at which it occurs.

The events at a process are linearly ordered by their order of occurrence. The execution of process pi produces a sequence of events e1i , e2i , , exi ,ex+1i , and is denoted by _i:

$$\mathcal{H}_i = (h_i, \rightarrow_i).$$

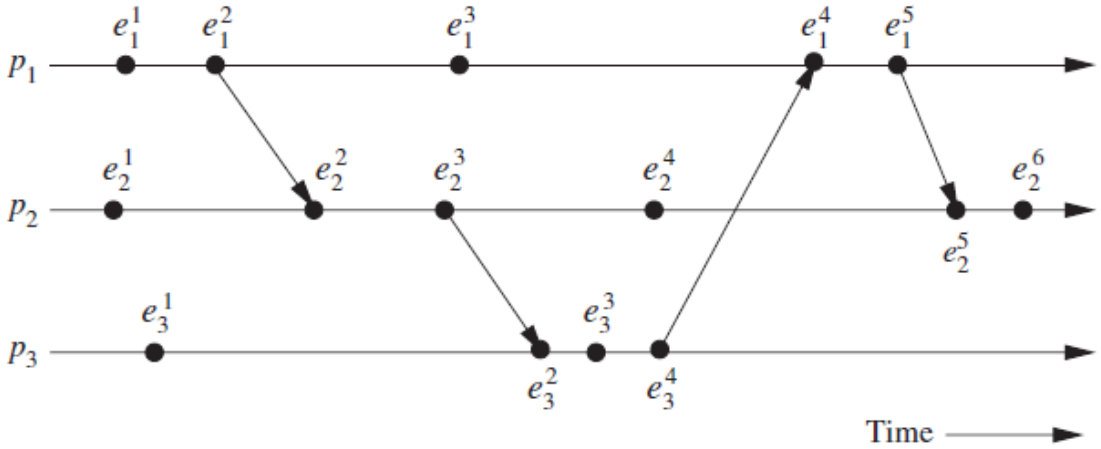
where hi is the set of events produced by pi and binary relation →i defines a linear order on these events. Relation →i expresses causal dependencies among the events of pi. The send and the receive events signify the flow of information between processes and establish causal dependency from the sender process to the receiver process. A relation →msg that captures the causal dependency due to message exchange, is defined as follows. For every message m that is exchanged between two processes, we have

$$send(m) \rightarrow_{msg} rec(m).$$

defines causal dependencies between the pairs of corresponding send and receive events.

The evolution of a distributed execution is depicted by a space–time diagram. Figure 2.1 shows the space–time diagram of a distributed execution involving three processes. A

horizontal line represents the progress of the



process; a dot indicates an event; a slant arrow indicates a message transfer. Generally, the execution of an event takes a finite amount of time; however, since we assume that an event execution is atomic (hence, indivisible and instantaneous), it is justified to denote it as a dot on a process line. In this figure, for process p1, the second event is a message send event, the third event is an internal event, and the fourth event is a message receive event.

Causal precedence relation

The execution of a distributed application results in a set of distributed events produced by the processes. Let H =∪ ihi denote the set of events executed in a distributed computation. Next, we define a binary relation on the set H, denoted as →, that expresses causal dependencies between events in the distributed execution.

$$\forall e_i^x,\ \forall e_j^y \in H,\quad e_i^x \rightarrow e_j^y \quad \Leftrightarrow \quad \begin{cases} e_i^x \rightarrow_i e_j^y \text{ i.e., } (i=j) \wedge (x<y) \\ \text{or} \\ e_i^x \rightarrow_{msg} e_j^y \\ \text{or} \\ \exists e_k^z \in H : e_i^x \rightarrow e_k^z \ \wedge\ e_k^z \rightarrow e_j^y \end{cases}$$

The causal precedence relation induces an irreflexive partial order on the events of a distributed computation [6] that is denoted as _=(H, →).

Logical vs. physical concurrency

In a distributed computation, two events are logically concurrent if and only if they do not causally affect each other. Physical concurrency, on the other hand, has a connotation that the events occur at the same instant in physical time. Note that two or more events may be logically concurrent even though they do not occur at the same instant in physical time. For

example, in Figure 2.1, events in the set {e3 1_ e4 2_ e3 3} are logically concurrent, but they occurred at different instants in physical time. However, note that if processor speed and message delays had been different, the execution of these events could have very well coincided in physical time. Whether a set of logically concurrent events coincide in the physical time or in what order in the physical time they occur does not change the outcome of the computation. Therefore, even though a set of logically concurrent events may not have occurred at the same instant in physical time, for all practical and theoretical purposes, we can assume that these events occured at the same instant in physical time.

2.3 Models of communication networks

There are several models of the service provided by communication networks, namely, FIFO (first-in, first-out), non-FIFO, and causal ordering. In the FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel. In the non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order. The "causal ordering" model [1] is based on Lamport's "happens before" relation. A system that supports the causal ordering model satisfies the following property:

$$\textbf{CO: For any two messages } m_{ij} \text{ and } m_{kj}, \text{ if } send(m_{ij}) \longrightarrow send(m_{kj}), \text{ then } rec(m_{ij}) \longrightarrow rec(m_{kj}).$$

That is, this property ensures that causally related messages destined to the same destination are delivered in an order that is consistent with their causality relation. Causally ordered delivery of messages implies FIFO message delivery. Furthermore, note that CO $\subset$ FIFO $\subset$ Non-FIFO. Causal ordering model is useful in developing distributed algorithms. Generally, it considerably simplifies the design of distributed algorithms because it provides a built-in synchronization. For example, in replicated database systems, it is important that every process responsible for updating a replica receives the updates in the same order to maintain database consistency. Without causal ordering, each update must be checked to ensure that database consistency is not being violated. Causal ordering eliminates the need for such checks.

2.4 Global state of a distributed system

The global state of a distributed system is a collection of the local states of its components, namely, the processes and the communication channels. The state of a process at any time is defined by the contents of processor registers, stacks, local memory, etc. and depends on the local context of the distributed application. The state of a channel is given by the set of messages in transit in the channel.

The occurrence of events changes the states of respective processes and channels, thus causing transitions in global system state. For example, an internal event changes the state of the process at which it occurs. A send event (or a receive event) changes the state of the process that sends (or receives) the message and the state of the channel on which the message is sent (or received).

Let LSx i denote the state of process pi after the occurrence of event ex i and before the event ex+1 i . LS0 i denotes the initial state of process pi. LSx i is a result of the execution of all the events executed by process pi till ex i . Let send(m)≤LSx i denote the fact that ∃ y:1≤y≤x :: ey i = send(m). Likewise, let rec(m)_≤LSx i denote the fact that ∀ y:1≤y≤x :: ey i_=rec(m). The state of a channel is difficult to state formally because a channel is a distributed entity and its state depends upon the states of the processes it connects. Let SCx_y ij denote the state of a channel Cij defined as follows:

$$SC_{ij}^{x,y} = \{m_{ij} \mid send(m_{ij}) \leq LS_i^x \wedge rec(m_{ij}) \not\leq LS_j^y\}.$$

Thus, channel state SCx_y ij denotes all messages that pi sent up to event ex i and which process pj had not received until event ey j .

2.4.1 Global state

The global state of a distributed system is a collection of the local states of the processes and the channels. Notationally, the global state GS is defined as

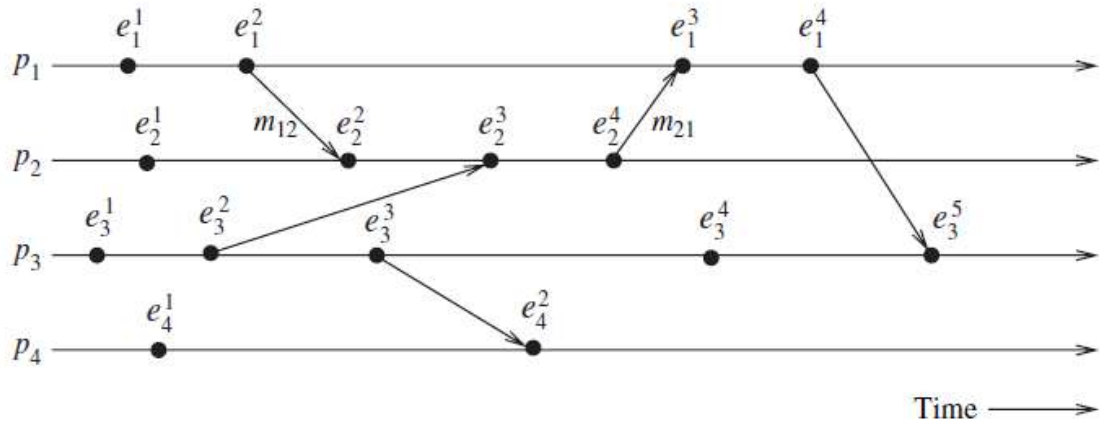
$$GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j,z_k}\}.$$

For a global snapshot to be meaningful, the states of all the components of the distributed system must be recorded at the same instant. This will be possible if the local clocks at processes were perfectly synchronized or there was a global system clock that could be instantaneously read by the processes. However, both are impossible. However, it turns out that even if the state of all the components in a distributed system has not been recorded at the same instant, such a state will be meaningful provided every message that is recorded as received is also recorded as sent. Basic idea is that an effect should not be present without its cause. A message cannot be received if it was not sent; that is, the state should not violate causality. Such states are called *consistent global states* and are meaningful global states. Inconsistent global states are not meaningful in the sense that a distributed system can never be in an inconsistent state.

A global state GS = $GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j,z_k}\}$ is a *consistent global state* iff it satisfies the following condition:

$$\forall m_{ij} : send(m_{ij}) \not\leq LS_i^{x_i} \Rightarrow m_{ij} \notin SC_{ij}^{x_i,y_j} \wedge rec(m_{ij}) \not\leq LS_j^{y_j})$$

That is, channel state SCyi_zk ik and process state LSzk k must not include any message that process pi sent after executing event exi i .



states $\{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$ is consistent; all the channels are empty except C_{21} that contains message m_{21}.

A global state $GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j,z_k}\}$ is *transitless* iff

$$\forall i, \forall j : 1 \leq i, j \leq n :: SC_{ij}^{y_i,z_j} = \phi.$$

Thus, all channels are recorded as empty in a transitless global state. A global state is *strongly consistent* iff it is transitless as well as consistent. Note that in Figure 2.2, the global state consisting of local states {LS21 , LS32 ,LS43 , LS24} is strongly consistent. Recording the global state of a distributed system is an important paradigm when one is interested in analyzing, monitoring, testing, or verifying properties of distributed applications, systems, and algorithms. Design of efficient methods for recording the global state of a distributed system is an important problem.

2.7 Models of process communications

There are two basic models of process communications – synchronous and asynchronous. The *synchronous* communication model is a blocking type where on a message send, the sender process blocks until the message has been received by the receiver process. The sender process resumes execution only after it learns that the receiver process has accepted the message. Thus, the sender and the receiver processes must synchronize to exchange a message.

On the other hand, *asynchronous* communication model is a non-blocking type where the sender and the receiver do not synchronize to exchange a message. After having sent a message, the sender process does not wait for the message to be delivered to the receiver process. The message is bufferred by the system and is delivered to the receiver process when it is ready to accept the message. A buffer overflow may occur if a process sends a large number of messages in a burst to another process.

Neither of the communication models is superior to the other. Asynchronous communication provides higher parallelism because the sender process can execute while the message is in

transit to the receiver. However, an implementation of asynchronous communication requires more complex buffer management.

In addition, due to higher degree of parallelism and non-determinism, it is much more difficult to design, verify, and implement distributed algorithms for asynchronous communications. The state space of such algorithms are likely to be much larger. Synchronous communication is simpler to handle and implement. However, due to frequent blocking, it is likely to have poor performance and is likely to be more prone to deadlocks.

Karpagam Academy of Higher Education
Department of Computer Applications
Subject Code/Name : Elective:17CAP505N/Distributed Computing

Objective Type Questions

	UNIT I							
S.No	**Questions**	**OPTION 1**	**OPTION 2**	**OPTION 3**	**OPTION 4**	**OPTION 5**	**OPTION 6**	**Answer Key**
1	A _____ is a collection of an independent entities that cooperate to solve a problem that cannot be individually solved	distributed system	network	component	device			**distributed system**
2	Distributed systems have been in existence since the start of the _____ .	world	universe	machine	states			**universe**
3	Typically the computers are _____ and are loosely coupled while they operate to address a problem collectively.	autonaomous	fully autonomous	semi autonomous	completed			**semi autonomous**
4	A collection of independent computers that appears to the users of the system as a single _____.	inherent computer	different devices	Iinherent devices	coherent computer			**coherent computer**
5	A wide range of computers from weakly coupled systems such as _____	wide area networks.	MAN	LAN	Networks			**wide area networks.**
6	_____ coupled systems such as local area networks.	weakly	Strongly	completely	terminated.			**Strongly**
7	No common _____ gives rise to the inherent asynchrony amongst the processors.	errors	terminal clock	physical clock	CMOS			**physical clock**
8	No_____ is a key feature the requires message passing for communication.	Distributed memory	unshared memory	fully distibuted memory	shared memory			**shared memory**
9	A DS may still provide the abstraction of a common address space via the distributed shared _____.	memory abstraction	deletion	enapsulation	data abstraction			**memory abstraction**
10	The geographically wider apart that the processors are the more representative is the system of a _____.	DC	DS	Memory coherence	Memory			**DS**
11	Recently, the network/cluster of a workstations configuration connecting processors on a _____ is also being increasingly regarded as a small DS.	WAN	MAN	LAN	Fully connected Network			**LAN**
12	The _____ search engine is based on the NOW architecture.	Yahoo	LYNX	Bing	Google			**Google**
13	The processors are_____ in that they have different speeds and each can be running a different OS.	tightly coupled	coupled	loosely coupled	no coupling			**loosely coupled**
14	Each computer has a memory processing unit and the computers are connected by a _____.	Distributed network	communication network	DS	Connection network			**communication network**
15	The distributed software is also termed as _____.	software	hardware	middleware	terminal			**middleware**
16	A _____ is the execution of processes across the DS to collaboratively achieve a common goal.	distributed execution	dustributed computers	distrinuted systems	connected executions			**distributed execution**
17	An _____ is also sometimes termed as computation or a run.	termination	execution	connection	suspension			**execution**

18	The DS uses a_____ architecture to break down the complexity of system design.	nonlayered	category	layered	boxed			**layered**
19	The _____ is the distributed software that drives the DS.	software	hardware	terminal	middleware			**middleware**
20	Various primitives and calls to functions define in various _____ of the middleware layer are embedded in the user program code.	libraries	files	records	sources			**libraries**
21	The _____ mechanism conceptually works like a local procedure call, with the difference that the procedure code may reside on the remote machine.	IP	RPC	TCP	UDP			**RPC**
22	Currently deployed _____ of middleware often use CORBA, DCOM, Java, and RMI technologies.	Un commercial versions	Monetarized	commercial versions	lost verisons			**commercial versions**
23	The message passing interface developed in the research community is an example of an _____ for various communication functions.	communications	network	resources	interface			**interface**
24	_____such as peripherals, complete data sets in databases, special libraries, as well as data cannot be full replicated at all the sites because it is often neither practical nor cost effective.	Resources	network	interfaces	communications			**Resources**
25	A DS has the inherent potential to provide _____reliability.	decreased	increased	tight	loose			**increased**
26	Reliability entails several aspects are availability, integrity and _____	unfault tolerance	tightly coupled	fault tolerance.	loosely coupled			**fault tolerance.**
27	By _____ and assessing geographically remote data and resources the performance/cost ratio is increased.	network sharing	file sharing	devcies sharing	resource sharing			**resource sharing**
28	Scalability as the processors are usually connected by WAN adding more processors does not pose a direct bottleneck for the _____	communication network.	network	addresses	devices			**communication network.**
29	A multiprocessor system is a parallel system in which the multiple processors have direct access to shared memory _____	address space	common address space.	memory space	space			**common address space.**
30	A _____ usually corresponds to a uniform memory access architecture in which the access latency.	uniprocessor system	mainframes	multiprocessor system	computers			**multiprocessor system**
31	The processors are usually of the same type and are housed within the same box,/container with a _____.	unshared memory	devices	computers	shared memory			**shared memory**
32	Various techniques such as buffering or more elaborate interconnection designs can_____.	address collisions	noncollision systems	noncollision devices	collisions addresses			**address collisions**
33	Observe that for the butterfly and the omega networks the paths from the different inputs to any one output form a _____.	travelling space	spanning tree	timimg limit	synchroniation			**spanning tree**

34	The processors are in close physical proximity and are usually very tightly coupled and are connected by an _____.	unidirectional	bidirectional	interconnection network	interfaces			**interconnection network**
35	A multicomputer system that has a common address space or via _____	sharing	detecting	avoiding	message passing			**message passing**
36	A _____ that has a common address space usually corresponds to a non uniform memory access architecture in which the latency to access various shared memory locations from the different processors varies.	single computer	multicomputer	resources	multicomputer system			**multicomputer system**
37	The regular and _____ have interesting mathematical properties that enable very easy routing and provide many rich features ash as alternate routing.	asymmetrical topologies	ring topology	symmetrical topologies	star topology			**symmetrical topologies**
38	The processors are labelled such that the shortest path between any two processors is the _____ between the processor labels.	unipolar	Hamming distance	bipolar	distance vector			**Hamming distance**
39	Routing in the hypercube is done by _____.	hop_by_hop	terminals	devices	systems			**hop_by_hop**
40	The 4D hypercube is formed by connecting the corresponding edges of two 3D hypercube along the_____.	third dimension	fourth dimension	single dimension	two dimension			**fourth dimension**
41	The hypercube and its variant topologies have very interesting mathematical properties with implications for _____.	fault tolerance	routing tolerance	routing and fault tolerance	nothing fault			**routing and fault tolerance**
42	_____belong to a class of parallel computers that are physically collocated are very tightly coupled and have a common system clock.	bit processors	memory processors	stack processors	Array processors			**Array processors**
43	The primary and most efficacious use of parallel systems is for obtaining a higher throughput by dividing the computational workload among the _____.	devices	terminals	covers	processors			**processors**
44	Searching through large state spaces can be performed with significant speedup on _____.	distributed machines	DS	parallel machines	DC			**parallel machines**
45	SISD mode corresponds to the conventional processing in the Von Neumann paradigm with a single CPU, and a single memory unit connected by a _____ .	Devices	terminals	processors	system bus			**system bus**
46	_____mode corresponds to the execution of different operations in parallel on the same data.	SISD	SIMD	MIMD	MISD			**MISD**
47	Sun Ultra servers, Multicomputer PCs and IBM SP machines are examples of machines that execute in _____.	SISD	SIMD	MIMD mode	MISD			**MIMD mode**
48	The _____ among a set of modules whether hardware or software is measured in terms of the interdependency and binding and or homogeneity among the modules.	cohesion	degree of coupling	tighlty coupled	loosely coupled			**degree of coupling**

49	_____arhitectures generally tend to be tightly coupled because of the common clocking of the shared instruction stream or the shared data stream.	SIMD and MIMD	SIMD	MIMD	MISD			**SIMD and MIMD**
50	The speedup depends on the number of processors and the mapping of the code to the_____.	terminals	processors	arrays	devices			**processors**
51	The ratio of the amount of computation to the amount of communication within the parallel. Distributed program is termed as _____	granularity.	complexity	durability	none of these			**granularity.**
52	The operating system running on loosely coupled processors which are themselves running loosely coupled software is classified as _____.	OS	network operating system	LINUX	DOS			**network operating system**
53	The operating system running on tightly coupled processors, which are themselves running themselves running tightly coupled software is classified as a _____	OS	DOS	multiprocessor operating system.	LINUX			**multiprocessor operating system.**
54	Implementing the abstraction has a certain cost but it simplifies the task of the _____.	system programmer	network programmer	programmer	application programmer			**application programmer**
55	_____ communication primitives are denoted Send() and Receive() respectively.	receives	sends	transmits	Message send and message receive			**Message send and message receive**
56	The buffered option which is a the standard option copies the data from the user buffer to the_____.	buffer	memory	kernel buffer	virtual memory			**kernel buffer**
57	A distributed application runs as a collection of processes on a _____.	DC	distributed system	terminals	fdevices			**distributed system**
58	In the_____, each channel acts as a first in first out message queue and thus message ordering is preserved by a channel.	FIFO model	LIFO model	System contribution	System testing			**FIFO model**
59	The state of a _____ is given by the set of messages in transit in the channel.	devices	channel	transmits	receives			**channel**
60	The occurrence of events changes the states of respective processes and channels thus causing transitions in _____.	local kernel	local systems	global system state	state devices			**global system state**

2.1 Message ordering and group communication

Inter-process communication via message-passing is at the core of any distributed system. In this chapter, we will study non-FIFO, FIFO, causal order, and synchronous order communication paradigms for ordering messages. We will then examine protocols that provide these message orders. We will also examine several semantics for group communication with multicast – in particular, causal ordering and total ordering. We will then look at how exact semantics can be specified for the expected behavior in the face of processor or link failures. Multicasts are required at the application layer when superimposed topologies or overlays are used, as well as at the lower layers of the protocol stack. We will examine some popular multicast algorithms at the network layer. An example of such an algorithm is the Steiner tree algorithm, which is useful for setting up multi-party teleconferencing and videoconferencing multicast sessions.

Notation

As before, we model the distributed system as a graph (N, L). The following notation is used to refer to messages and events:

• When referring to a message without regard for the identity of the sender and receiver processes, we use mi. For message mi, its send and receive events are denoted as s^i and r^i, respectively.

• More generally, send and receive events are denoted simply as s and r. When the relationship between the message and its send and receive events is to be stressed, we also use M, send(M), and receive(M), respectively.

For any two events a and b, where each can be either a send event or a receive event, the notation $a \sim b$ denotes that a and b occur at the same process, i.e., $a \in E_i$ and $b \in E_i$ for some process i. The send and receive event pair for a message is said to be a pair of *corresponding* events. The send event corresponds to the receive event, and vice-versa. For a given execution E, let the set of all send–receive event pairs be denoted as $T = \{(s, r) \in E_i \times E_j \mid s \text{ corresponds to } r\}$. When dealing with message ordering definitions, we will consider only send and receive events, but not internal events, because only communication events are relevant.

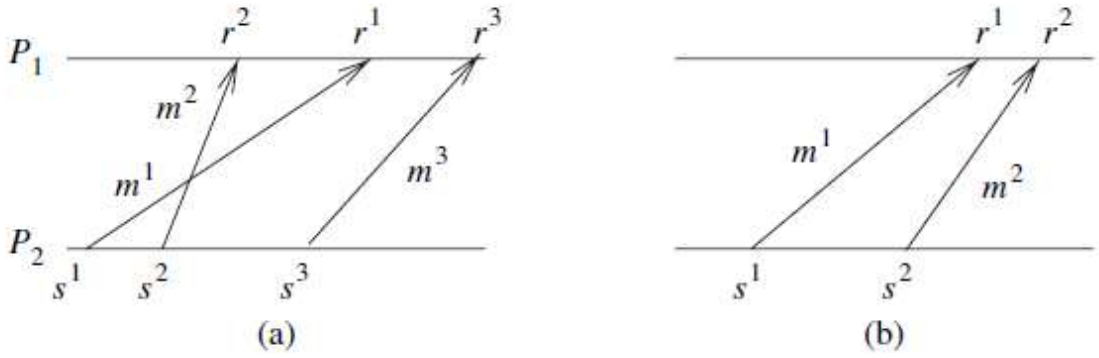
2.2 Message ordering paradigms

The order of delivery of messages in a distributed system is an important aspect of system executions because it determines the messaging behavior that can be expected by the distributed program. Distributed program logic greatly depends on this order of delivery. To simplify the task of the programmer, programming languages in conjunction with the middleware provide certain well-defined message delivery behavior. The programmer can then code the program logic with respect to this behavior.

Several orderings on messages have been defined: (i) non-FIFO, (ii) FIFO, (iii) causal order, and (iv) synchronous order. There is a natural hierarchy among these orderings. This hierarchy represents a trade-off between concurrency and ease of use and implementation. After studying the definitions of and the hierarchy among the ordering models, we will study some implementations of these orderings in the middleware layer.

2.1.1 Asynchronous executions

Definition (A-execution) An asynchronous execution (or A-execution) is an execution (E,γ) for which the causality relation is a partial order. There cannot exist any causality cycles in any real asynchronous execution because cycles lead to the absurdity that an event causes itself. On any logical link between two nodes in the system, messages may be delivered in any order, *not necessarily* first-in first-out. Such executions are also known as *non-FIFO executions*. Although each physical link typically delivers the messages sent on it in FIFO order due to the physical properties of the medium, a logical link may be formed as a composite of physical links and multiple paths may exist between the two end points of the logical link. As an example, the mode of ordering at the Network Layer in connectionless networks such as IPv4 is non-FIFO. Figure 2.1(a) illustrates an A-execution under non-FIFO ordering.



2.1.2 FIFO executions

Definition (FIFO executions) A FIFO execution is an A-execution in which, for all

$$(s, r) \text{ and } (s', r') \in \mathcal{T}, (s \sim s' \text{ and } r \sim r' \text{ and } s \prec s') \Longrightarrow r \prec r'.$$

On any logical link in the system, messages are necessarily delivered in the order in which they are sent. Although the logical link is inherently non- FIFO, most network protocols provide a connection-oriented service at the transport layer. Therefore, FIFO logical channels can be realistically assumed when designing distributed algorithms. A simple algorithm to implement a FIFO logical channel over a non-FIFO channel would use a separate numbering scheme to sequence the messages on each logical channel. The sender assigns and appends a (*sequence_num, connection_id*) tuple to each message.

The receiver uses a buffer to order the incoming messages as per the sender's sequence numbers, and accepts only the "next" message in sequence.

2.1.3 Causally ordered (CO) executions

Definition (Causal order (CO)) A CO execution is an A-execution in which,

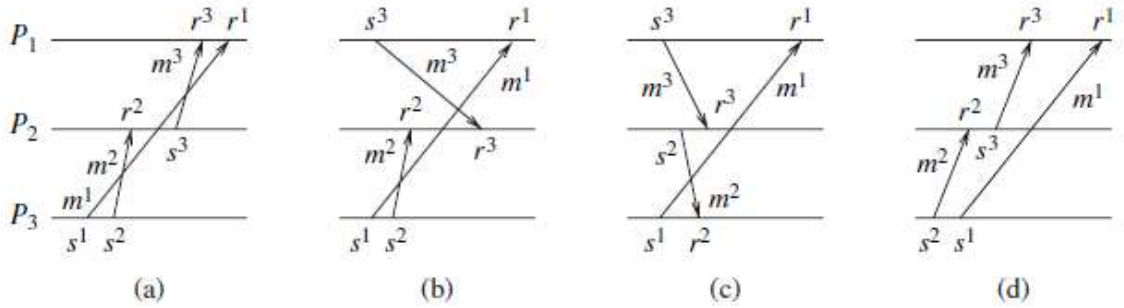
$$\text{for all } (s, r) \text{ and } (s', r') \in \mathcal{T}, (r \sim r' \text{ and } s \prec s') \Longrightarrow r \prec r'.$$

If two send events s and s_ are related by causality ordering (not physical time ordering), then a causally ordered execution requires that their corresponding receive events r and r_ occur in the same order at all common destinations. Note that if s and s_ are not related by causality, then CO is vacuously satisfied because the antecedent of the implication is false.

Examples

• **Figure 2(a)** shows an execution that violates CO because s1 γ s3 and at the common destination P1, we have r3 γ r1.

• **Figure 2(b)** shows an execution that satisfies CO. Only s1 and s2 are related by causality but the destinations of the corresponding messages are different.



• **Figure 2(c)** shows an execution that satisfies CO. No send events are related by causality.

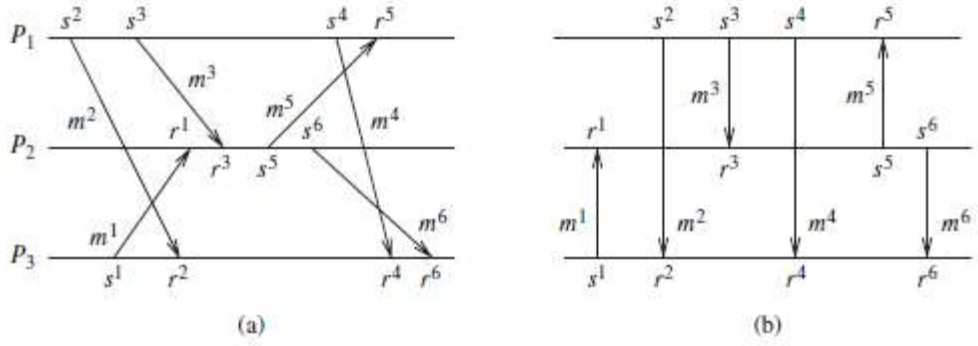
• **Figure 2(d)** shows an execution that satisfies CO. s2 and s1 are relatez by causality but the destinations of the corresponding messages are different. Similarly for s2 and s3. Causal order is useful for applications requiring updates to shared data, implementing distributed shared memory, and fair resource allocation such as granting of requests for distributed mutual exclusion.

To implement CO, we distinguish between the arrival of a message and its delivery. A message m that arrives in the local OS buffer at Pi may have to be delayed until the messages that were sent to Pi causally before m was sent (the "overtaken" messages) have arrived and are processed by the application. The delayed message m is then given to the application for processing. The event of an application processing an arrived message is referred to as a *delivery* event (instead of as a *receive* event) for emphasis.

2.1.4 Synchronous execution (SYNC)

When all the communication between pairs of processes uses synchronous send and receive primitives, the resulting order is the synchronous order. As each synchronous communication involves a handshake between the receiver and the sender, the corresponding send and receive events can be viewed as occurring instantaneously and atomically. In a timing diagram, the "instantaneous" message communication can be shown by bidirectional vertical message lines. Figure 3(a) shows a synchronous execution on an asynchronous system. Figure 3(b) shows the equivalent timing diagram with the corresponding instantaneous message communication.

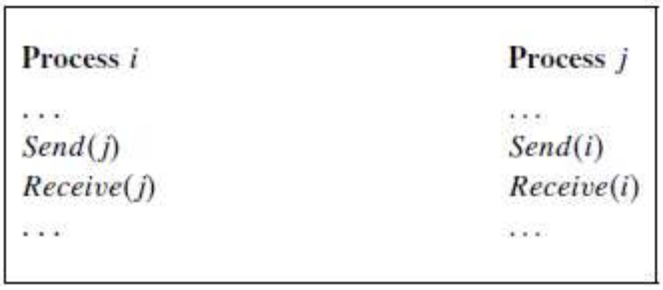
The "instantaneous communication" property of synchronous executions requires a modified definition of the causality relation because for each (s, r)∈ *T*, the send event is not causally ordered before the receive event. The two events are viewed as being atomic and simultaneous, and neither event precedes the other.

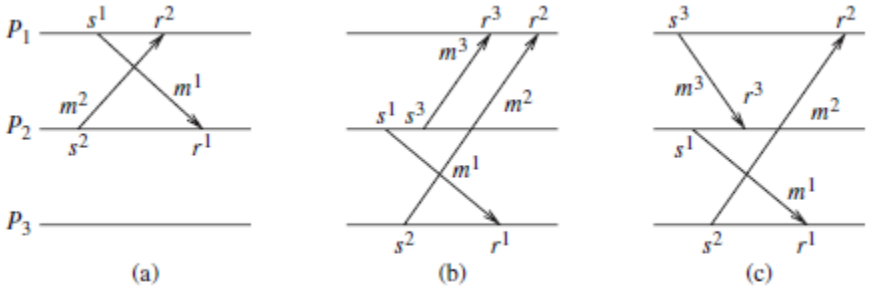


2.3 Asynchronous execution with synchronous communication

When all the communication between pairs of processes is by using synchronous send and receive primitives, the resulting order is synchronous order. The send and receive events of a message appear instantaneous, see the example in Figure 3. We now address the following question

• If a program is written for an asynchronous system, say a FIFO system, will it still execute correctly if the communication is done by synchronous primitives instead? There is a possibility that the program may *deadlock*, as shown by the code in Figure 4. Charron-Bost *et al.* observed that a distributed algorithm designed to run correctly on asynchronous systems (called *A-executions*) may not run correctly on synchronous systems. An algorithm that runs on an asynchronous system may *deadlock* on a synchronous system.





2.3.1 Executions realizable with synchronous communication (RSC)

An execution can be modeled (using the interleaving model) as a feasible schedule of the events to give a total order that extends the partial order (E,γ). In an A-execution, the messages can be made to appear instantaneous if there exists a linear extension of the execution, such that each send event is immediately followed by its corresponding receive event in this linear extension. Such an A-execution can be realized under synchronous communication and is called a *realizable with synchronous communication* (RSC) execution.

2.3.2 Hierarchy of ordering paradigms

Let *SYNC* (or *RSC*), *CO*, *FIFO*, and *A* denote the set of all possible executions ordered by synchronous order, causal order, FIFO order, and non- FIFO order, respectively. We have the following results:

• For an A-execution, A is RSC if and only if A is an S-execution.

• $RSC \subset CO \subset FIFO \subset A$. This hierarchy is illustrated in Figure 6.(a), and example executions of each class are shown side-by-side in Figure 6.(b). Figure 6.(a) shows an execution that belongs to *A* but not to *FIFO*. Figure 6.(a) shows an execution that belongs to *FIFO* but not to *CO*. Figures 6.(b) and (c) show executions that belong to *CO* but not to *RSC*.

• The above hierarchy implies that some executions belonging to a class X will not belong to any of the classes included in X. Thus, there are more restrictions on the possible message orderings in the smaller classes. Hence, we informally say that the included classes have less concurrency. The degree of concurrency is most in *A* and least in *SYNC*.

• A program using synchronous communication is easiest to develop and verify. A program using non-FIFO communication, resulting in an A execution, is hardest to design and verify. This is because synchronous order offers the most simplicity due to the restricted number of possibilities, whereas non-FIFO order offers the greatest difficulties because it admits a much larger set of possibilities that the developer and verifier need to account for. Thus, there is an inherent trade-off between the amount of concurrency provided, and the ease of designing and verifying distributed programs.

2.4 Classification of application-level multicast algorithms

We have seen some algorithmically challenging techniques in the design of multicast algorithms. The most general scenario allows each process to multicast to an arbitrary and dynamically changing group of processes at each step. As this generality incurs more overhead, algorithms implemented on real systems tend to be more “centralized” in one sense or another: Defago *et al.* give an exhaustive survey and this section is based on this survey. For details of the various protocols, please refer to the survey. Many multicast protocols have been developed and deployed, but they can all be classified as belonging to one of the following five classes.

Communication history-based algorithms

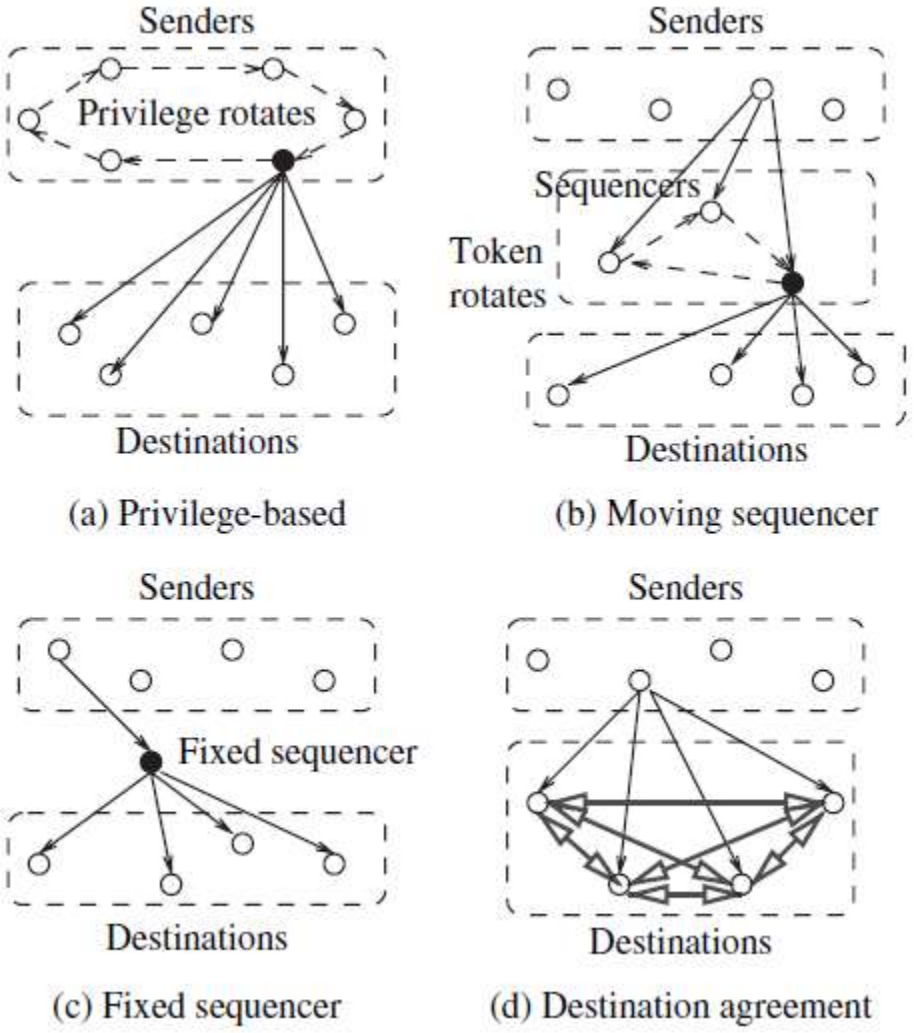
Algorithms in this class use a part of the communication history to guarantee ordering requirements. The RST and KS algorithms belong to this class, and provide only causal ordering. They do not need to track separate groups, and hence work for open-group multicasts. Lamport’s algorithm, wherein messages are assigned scalar timestamps and a process can deliver a message only when it knows that no other message with a lower

timestamp can be multicast, also belongs to this class. The NewTop protocol, which extends Lamport's algorithm to overlapping groups, also guarantees both total and causal ordering. Both these algorithms use closed group configurations.

Privilege-based algorithms

The operation of such algorithms is illustrated in Figure 6.(a). A token circulates among the sender processes. The token carries the sequence number for the next message to be multicast, and only the token-holder can multicast. After a multicast send event, the sequence number is updated. Destination processes deliver messages in the order of increasing sequence numbers. Senders need to know the other senders, hence closed groups are assumed. Such algorithms can provide total ordering, as well as causal ordering using a closed group configuration.

Examples of specific algorithms are On-Demand, and Totem. They differ in implementation details such as whether a token ring topology is assumed



(Totem) or not (On-Demand). Such algorithms are not scalable because they do not permit concurrent send events. Hence they are of limited use in large systems.

Moving sequencer algorithms

The operation of such algorithms is illustrated in Figure 6.(b). The original algorithm was proposed by Chang and Maxemchuck; various variants of it were given by the Pinwheel and RMP algorithms. These algorithms work as follows.

(1) To multicast a message, the sender sends the message to all the sequencers.

(2) Sequencers circulate a token among themselves. The token carries a sequence number and a list of all the messages for which a sequence number has already been assigned – such messages have been sent already.

(3) When a sequencer receives the token, it assigns a sequence number to all received but unsequenced messages. It then sends the newly sequenced messages to the destinations, inserts these messages in to the token list, and passes the token to the next sequencer.

(4) Destination processes deliver the messages received in the order of increasing sequence number. Moving sequencer algorithms guarantee total ordering.

Fixed sequencer algorithms

The operation of such algorithms is illustrated in Figure 6.(c). This class is a simplified version of the previous class. There is a single sequencer (unless a failure occurs), which makes this class of algorithms essentially centralized. The propagation tree approach studied earlier, belongs to this class. Other algorithms are the ISIS sequencer, Amoeba, Phoenix, and Newtop's asymmetric algorithm. Let us look briefly at Newtop's asymmetric algorithm.

All processes maintain logical clocks, and each group has an independent sequencer. The unicast from the sender to the sequencer, as well as the multicast from the sequencer are timestamped. A process that belongs to multiple groups must delay the sending of the next message (to the relevant sequencer) until it has received and processed all messages, from the various sequencers, corresponding to the previous messages it sent. Assuming FIFO channels, it can be shown that total order is maintained.

Destination agreement algorithms

The operation of such algorithms is illustrated in Figure 6.(d). In this class of algorithms, the destinations receive the messages with some limited ordering information. They then exchange information among themselves to define an order. There are two sub-classes here: (i) the first sub-class uses timestamps (Lamport's three-phase belongs to this sub-class); (ii) the second sub-class uses an agreement or "consensus" protocol among the processes.

2.5 Distributed multicast algorithms at the network layer

Several applications can interface directly with the network layer and the lower hardware-related layers to exploit the physical connectivity and the physical topology for group communication. The network is viewed as a graph (N, L), and various graph algorithms – centralized or distributed – are run to establish and maintain efficient routing structures. For example,

• LANs connected by bridges maintain spanning trees for distributing information and for forward/backward learning of destinations;

• the network layer of the Internet has a rich suite of multicast algorithms. In this section, we will study the principles underlying several such algorithms. Some of the algorithms in this

section may not be distributed. Nevertheless, they are intended for a distributed setting, namely the LAN or the WAN.

2.5.1 Reverse path forwarding (RPF) for constrained flooding

Broadcasting data using flooding in a network (N,L) requires up to 2|L| messages. Reverse path forwarding (RPF) is a simple but elegant technique that brings down the overhead significantly at very little cost. Network nodes are assumed to run the distance vector routing (DVR) algorithm, which was used in the Internet until 1983. (Since 1983, the LSR-based algorithms have been used. These are more sophisticated and provide more information than that required by DVR.) The simple DVR algorithm assumes that each node knows the next hop on the path to each destination x. This path is assumed to be the approximation to the "best" path. Let Next_hop(x) denote the function that gives the next hop on the "best" path to x. The RPF algorithm leverages the DVR algorithm for point-to-point routing, to achieve constrained flooding. The RPF algorithm for constrained flooding is shown in the following Algorithm.

Algorithm Reverse path forwarding (RPF).
(1) When process Pi wants to multicast message M to group Dests:
(1a) **send** M(i, Dests) on all outgoing links.
(2) When a node i receives message M(x, Dests) from node j:
(2a) **if** Next_hop(x) = j **then** // this will necessarily be a new message
(2b) **forward** M(x, Dests) on all other incident links besides (i,j);
(2c) **else** ignore the message.

This simple RPF algorithm has been experimentally shown to be effective in bringing the number of messages for a multicast closer to |N| than to |L|. Actually, the algorithm does a broadcast to all the nodes, and this broadcast is smartly curtailed to approximate a spanning tree. The curtailed broadcast is effective because, implicitly, an approximation to a tree rooted at the source is identified, without it being computed or stored at any node.
Pruning of the implicit broadcast tree can be used to deal with unwanted multicast packets. If a node receives the packets but the application running on it does not need the packets, and all "downstream" (in the implicit tree) nodes also do not need the packets, the node can send a *prune* message to the parent in the tree indicating that packets should not be forwarded on that edge. Implementing this in a dynamic network where the tree periodically changes and the application's node membership also changes dynamically is somewhat.

2.5.2 Steiner trees

The problem of finding an optimal "spanning" tree that spans only all nodes participating in a multicast group, known as the *Steiner tree problem*, is formalized as follows.

Steiner tree problem

Given a weighted graph (N,L) and a subset N' $\subseteq$ N, identify a subset L'$\subseteq$L such that (N',L') is a subgraph of (N,L) that connects all the nodes of N'. A *minimal Steiner tree* is a minimal-

weight subgraph (N',L'). The minimal Steiner tree problem has been well-studied and is known to be NP-complete. When the link weights change, the tree has to be recomputed to obtain the new minimal Steiner tree, making it even more difficult to use in dynamic networks.

Several heuristics have been proposed to construct an approximation to the minimal Steiner tree. A simple heuristic constructs a MST, and deletes edges that are not necessary. This algorithm is given by the first three steps of Algorithm. The worst case cost of this heuristic is twice the cost of the optimal solution. Algorithm can show better performance when using the heuristic by Kou *et al.*, given by steps 4 and 5 in the algorithm. The resulting Steiner tree cost is also at most twice the cost of the minimal Steiner tree, but behaves better on average.

Algorithm The Kou–Markowsky–Berman heuristic for a minimum Steiner tree.

Input: weighted graph $G = (N, L)$, and $N' \subseteq N$, where N' is the set of Steiner points

(1) Construct the complete undirected distance graph $G' = (N', L')$ as follows:
$L' = \{(v_i, v_j) \mid v_i, v_j \text{ in } N'\}$, and $wt(v_i, v_j)$ is the length of the shortest path from v_i to v_j in (N, L).
(2) Let T' be the minimal spanning tree of G'. If there are multiple minimum spanning trees, select one randomly.
(3) Construct a subgraph G_s of G by replacing each edge of the MST T' of G', by its corresponding shortest path in G. If there are multiple shortest paths, select one randomly.
(4) Find the minimum spanning tree T_s of G_s. If there are multiple minimum spanning trees, select one randomly.
(5) Using T_s, delete edges as necessary so that all the leaves are the Steiner points N'. The resulting tree, $T_{Steiner}$, is the heuristic's solution.

2.5.3 Multicast cost functions

Consider a source node s that has to do a multicast to Steiner nodes. As before, we are given the weighted graph (N,L) and the Steiner node set N'. For example, let cost(i) be the cost of the path from s to i in the routing scheme R. The *destination cost of* R is defined as

$$\frac{1}{|N'|} \sum_{i \in N'} cost(i)$$

. This represents the average cost of the routing. If the cost is measured in time delay, this routing function metric gives the shortest average time for the multicast to reach nodes in N'. As a variant, a link is counted only once even if it is used on the minimum cost path to multiple destinations. This variant reduces to the Steiner tree. The sum of the costs of the edges in the Steiner tree routing scheme R is defined as the *network cost*.

2.5.4 Delay-bounded Steiner trees

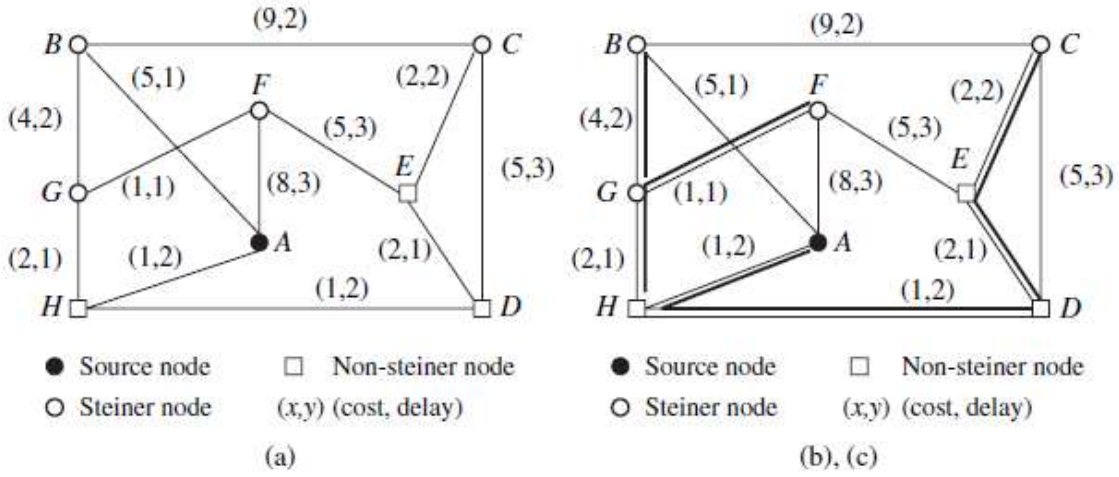
Multimedia networks and interactive applications have given rise to the need for a minimum Steiner tree that also satisfies delay constraints on the transmission. Thus now, the goal is not only to minimize the cost of the tree (measured in terms of a parameter such as the link weight, which models the available bandwidth or a similar cost measure) but also to minimize the delay (propagation delay). The problem is formalized as follows.

Delay-bounded minimal Steiner tree problem

Given a weighted graph (N, L), there are two weight functions *C*(l) and *D*(l) for each edge in L. *C*(l) is a positive real cost function on l ∈ L and *D*(l) is a positive integer delay function on l ∈L. For a given delay tolerance Δ, a given source s and a destination set Dest, where s_∪Dest = N'C N, identify a spanning tree T covering all the nodes in N', subject to the constraints below. Here, we let *path*_s_ v_ denote the path from s to v in T.

- $\sum_{l \in T} \mathcal{C}(l)$ is minimized, subject to
- $\forall v \in N', \sum_{l \in path(s,v)} \mathcal{D}(l) < \Delta.$

Finding such a minimal Steiner tree, subject to another parameter, is at least as difficult as finding a Steiner tree. It can be shown that this problem reduces to the Steiner tree problem. A detailed study of two heuristics to solve this problem is presented by Kompella *et al.*. A *constrained cheapest path* between x and y is the cheapest path between x and y that has delay less than _. The cost and delay on such a path are denoted by *C*(x,y) and *D(x,y)*, respectively. If two or more paths have the lowest cost, the lowest delay path is chosen. The steps to compute the constrained Steiner tree are shown in Algorithm. Step 1 computes the complete closure graph G' on nodes in N'. The two heuristics given below are used in Step 2 to greedily build a constrained Steiner tree on G'. Step 3 expands the tree edges in G' to their original paths in G. An example of a constrained Steiner



$\mathcal{C}(l)$ // cost of edge l
$\mathcal{D}(l)$ // delay of edge l
T; // constrained spanning tree to be constructed
$\mathcal{P}_C(x, y)$; // cost of constrained cheapest path from x to y
$\mathcal{P}_D(x, y)$; // delay on constrained cheapest path from x to y
$\mathcal{C}_d(x, y)$; // cost of the cheapest path with delay exactly d

Input: weighted graph $G = (N, L)$, and $N' \subseteq N$, where N' is the set of Steiner points, source is s, and Δ is the constraint on the delay.

1. Compute the closure graph G' on (N', L), to be the complete graph on N'. The closure graph is computed using the all-pairs constrained cheapest paths using a dynamic programming approach analogous to Floyd's algorithm. For any pair of nodes $x, y \in N'$:

   - $\mathcal{P}_C(x, y) = min_{d<\Delta}\mathcal{C}_d(x, y)$. This selects the cheapest constrained path, satisfying the condition of Δ, among the various paths possible between x and y. The various $\mathcal{C}_d(x, y)$ can be calculated using DP as follows:
   - $\mathcal{C}_d(x, y) = min_{z \in N}\{\mathcal{C}_{d-\mathcal{D}(z,y)}(x, z) + \mathcal{C}(z, y)\}$. For a candidate path from x to y passing through z, the path with weight exactly d must have a delay of $d - \mathcal{D}(z, y)$ for x to z when the edge (z, y) has delay $\mathcal{D}(z, y)$.

   In this manner, the complete closure graph G' is computed. $\mathcal{P}_D(x, y)$ is the delay on the constrained cheapest path that corresponds to a cost of $\mathcal{P}_C(x, y)$.
2. Construct a constrained spanning tree of G' using a greedy approach that sequentially adds edges to the subtree of the constrained spanning tree T (thus far) until all the Steiner points are included. The initial value of T is the singleton s. Consider that node u is in the tree and we are considering whether to add edge (u, v).

The following two edge selection criteria (heuristics) can be used to decide whether to include edge (u, v) in the tree:

- CST_{CD}: $f_{CD}(u,v) = \begin{cases} \dfrac{\mathcal{C}(u,v)}{\Delta - (\mathcal{P}_D(s,u) + \mathcal{D}(u,v))}, & \text{if } \mathcal{P}_D(s,u) + \mathcal{D}(u,v) < \Delta \\ \infty, & \text{otherwise.} \end{cases}$

  The numerator is the "incremental cost" of adding (u, v) and the denominator is the "residual delay" that could be afforded. The goal is to minimize the incremental cost, while also maximizing the residual delay by choosing an edge that has low delay. Thus, the heuristic picks the neighbor v that minimizes f_{CD}, for all u in T and all v adjacent to T.

- CST_C: $f_c = \begin{cases} \mathcal{C}(u,v), & \text{if } \mathcal{P}_D(s,u) + \mathcal{D}(u,v) < \Delta \\ \infty, & \text{otherwise.} \end{cases}$

  This heuristic picks the lowest cost edge between the already included tree edges and their nearest neighbor, as long as the total delay is less than Δ.

The chosen node v is included in T. This step 2 is repeated until T includes all $|N'|$ nodes in G'.

3. Expand the edges of the constrained spanning tree T on G' into the constrained cheapest paths they represent in the original graph G. Delete/break any loops introduced by this expansion.

• **Heuristic** CST_{CD} This heuristic tries to choose low-cost edges, while also trying to pick edges that maximize the remaining allowable delay. The motivation is to try to reduce the tree cost by path sharing, by extending the path beyond the selected edge. This heuristic has the tendency to optimize on delay also, while adding to the cost.

• **Heuristic** CST_C This heuristic simply minimizes the cost while ensuring that the delay bound is met.

Complexity Assuming integer-valued Δ, step 1, which finds the constrained cheapest shortest paths over all the nodes, has $O(n^3\Delta)$ time complexity. This is because all pairs of end and intermediate nodes have to be examined, for all integer delay values from 1 to Δ. Step 2, which constructs the constrained MST on the closure graph having k nodes, has $O(k^3)$ time complexity. Step 3, which expands the constrained spanning tree, involves expanding the k edges to up to n−1 edges each and then eliminating loops. This has O(kn) time overhead. The dominating step is step 1.

2.3.5 Core-based trees

In the core-based tree approach, each group has a center node, or *core* node. A multicast tree is constructed dynamically, and grows on-demand, as follows. (i) A node wishing to join the tree as a receiver sends a unicast "join" message to the core node. (ii) The join message marks the edges as it travels; it either reaches the core node, or some node which is already a part of the multicast tree. The path followed by the "join" message from its source till the core/multicast tree is grafted to the multicast tree, and defines the path to the "core." (iii) A node on the tree multicasts a message by using a flooding on the core tree. (iv) A node not on the tree sends a message towards the core node; as soon as the message reaches any node on the tree, the message is flooded on the tree. In a network with a dynamically changing

topology, care needs to be taken to maintain the tree structure and prevent messages from looping. This problem also exists for normal routing algorithms, such as the LSR and DVR algorithms, in dynamic networks. Current systems do not widely implement the Steiner tree for group multicast, even though it is more efficient after the initial cost to construct the Steiner tree. They prefer the simpler core-based tree (CBT) approach. Core-based trees have various variants. A multi-core-based tree has more than one core node. For all CBT algorithms, high-bandwidth links can be specially chosen over others for forming the tree. Core-based trees have a natural analog in wireless networks, wherein it is reasonable to constitute the core tree of high-bandwidth wired links or high-power wireless links.

Karpagam Academy of Higher Education
Department of Computer Applications
Subject Code/Name : Elective:17CAP505N/Distributed Computing

Objective Type Questions

	UNIT 2							
S.No	**Questions**	**OPTION 1**	**OPTION 2**	**OPTION 3**	**OPTION 4**	**OPTION 5**	**OPTION 6**	**Answer Key**
1	Interprocess communication via _____ is at the core of any distributed system.	message passing	message terminating	messages	termination detection			**message passing**
2	Multicasts are required at the application layer when superimposed topologies or overlays are used as well as at the lower layers of the _____.	stack	protocol stack	network protocol	layers			**protocol stack**
3	The order of delivery of _____ in a distributed system is an important aspect of system executions because it determines the messaging behaviour that can be expected by the DS.	stacks	queue	messages	terminals			**messages**
4	_____ logic greatly depends on this order of delivery.	DC	DS	distributed terminals	Distributed program			**Distributed program**
5	To simplify the task of the programmer, _____ in conjunction with the middleware provide certain well_defined message delivery behaviour.	programming languages	system programs	terminals	devices			**programming languages**
6	The _____ can then code the program logic with respect to this behaviour.	locker	programmer	terminating process	system locking			**programmer**
7	An _____ is an execution for which the causality relation is a partial order.	asynchronous	synchronous	synchronous execution	execution			**synchronous execution**
8	There cannot exist any causality _____ in any real asynchronous execution because cycles lead to the absurdity that an event causes itself.	stacks	queues	buffers	cycles			**cycles**
9	On any logical link between two nodes in the system, messages may be delivered in any order, not necessarily first in first out. Such executions are also known as _____	FIFO	LIFO model	NON LIFO	non FIFO			**non FIFO**
10	As an example, the mode of ordering at the network layer in connectionless networks such as _____ is non FIFO.	IPv1	IPv2	IPv3	IPv4			**IPv4**
11	Casual order is useful for applications requiring updates to shared data, implementing distributed shared memory, and fear resource allocation such as_____ of requests.	accessing	termianting	granting	accepting			**granting**
12	To implement CO, distinguish between the arrival of a _____and its delivery.	stacks	message	pototcols	terminals			**message**
13	When all the communication between pairs of processes uses synchronous send and receive primitives the resulting order is the_____.	synchronous order	asynchronous	synchronous	timing signals			**synchronous order**

14	The instantaneous_____ property of synchronous executions requires a modified definition of the causality relation.	interfaces	communication	terminals	resurces			**communication**
15	In a timing diagram the instaneous message communication can be shown by _____ vertical message lines.	unidirectional	unipolar	bidirectional	bipolar			**bidirectional**
16	If a program is written for an asynchronous system say a_____wilt still execute correctly if the communication is done by synchronous primitives instead?	LIFO SYSTEM	LIFO model	FIFO MODEL	FIFO system			**FIFO system**
17	An algorithm that runs on as asynchronous system may_____on a synchronous system.	lacking resources	feasible schedule	lacking communication	deadlock			**deadlock**
18	An execution can be modelled as a _____ of the events to give a total order.	unfeasible schedule	termination	feasible schedule	starting			**feasible schedule**
19	In an A_execution the messages can be made to appear instantaneous if there exists a linear extension of the execution such that each send event is immediately followed by its corresponding receive event.	unlinear	linear extension	unlinear extensions	linear			**linear extension**
20	A_execution can be realized under synchronous communication and is called a realizable with synchronous communication _____.	termination	separation	execution	disseparated			**execution**
21	In the _____ linear extension if the adjacent send event and its corresponding receive event are viewed atomically then that pair of events shares a common past and a common future with each other.	separated	communicated	non communicated	non_separated			**non_separated**
22	A characterization of the execution in terms of a graph structure called a____.	stack	terminal	crown	backlog			**crown**
23	The crown leads to a feasible test for a _____.	EXE	TXT	RPC	RSC execution			**RSC execution**
24	Intuitively the _____dependencies in a crown indicate that it is not possible to find a linear extension.	cyclic	acyclic	termination	detection			**cyclic**
25	The crown criterion states that an A_computation is _____.	RPC	RSC	TXT	EXE			**RSC**
26	A program using synchronous communication is easiest to develop and _____.	unverified	stabled	verify	unstabled			**verify**
27	A program using _____ communication resulting in a execution is hardest to design and verify.	LIFO SYSTEM	FIFO SYSTEM	Non LIFO	NonFIFO			**NonFIFO**
28	The events in _____ are scheduled as per some nonseparated linear extension and adjacent (s,r) events in this linear extension are executed sequentially in the synchronous system.	EXE	TXT	RPC	RSC execution			**RSC execution**
29	The partial order of the asynchronous execution remains_____ .	changed	distributed	system contribution	unchanged			**unchanged**
30	A (valid) S_execution can be trivially realized on an asynchronous system by scheduling the_____ in the order in which they appear in the S_execution.	terminals	stacks	messages	protocols			**messages**

31	The partial order of the S_execution remains unchanged but the communication occurs on an asynchronous system that uses_____ communication primitives.	synchronous	asynchronous	timimg limit	signals			**asynchronous**
32	Once a message send event is scheduled the middleware layer waits for an acknowledgement after the ack is received the synchronous send _____completes.	primitive	non primitive	synchronization variable	asychronous variable			**primitive**
33	There do not exist _____ with instantaneous communication that allows for synchronous communication to be naturally realized.	real systems	embedded systems	batch processing	timing systems			**real systems**
34	We need to _____ the basic question of how a system with synchronous communication can be implemented.	stacks	address	spaces	memory			**address**
35	We first examine non determinism in program execution and CSP as a representative synchronous programming language before examining an implementation of _____.	asynchronous communication	timing signals	synchronous communication	none of these			**synchronous communication**
36	Some algorithmically challenging techniques in the design of the _____	unicast algorithms	algorithms	procedures	multicast algorithms.			**multicast algorithms.**
37	The most general scenario allows each process to multicast to an arbitrary and _____ changing group of processes at each step.	statically	processing	timing	dynamically			**dynamically**
38	_____ in the class use a part of the communication history to guarantee ordering requirements.	procedures	stack processing	Algorithms	tokens			**Algorithms**
39	The _____ which extends Lamport's algorithm to overlapping groups also gurantees both total and casual ordering.	protocol	NewTop protocol	layered protocol	toping layers			**NewTop protocol**
40	A _____ circulates among the sender processes.	token	parsers	packets	terminals			**token**
41	The token carries the_____ for the next message to be multicast and only the token holder can multicast.	number	sequence number	unordered sequence	ordered sequence			**sequence number**
42	After a multicast send event the sequence number is _____.	deleted	connected	updated	disconnected			**updated**
43	_____ processes deliver messages in the order of increasing sequence numbers.	source	terminals	packets	Destination			**Destination**
44	_____need to know the other senders hence closed groups are assumed.	receivers	transmitters	receivers	Senders			**Senders**
45	All _____ can provide total ordering as well as causal ordering using a closed group configuration.	procedures	terminating process	detection process	algorithms			**algorithms**
46	_____ wherein messages are assigned scalar timestamps and a process can deliver a message only when it knows that no other message with a lower timestamp can be multicast also belongs to the class.	Lamport's algorithm	Pass algorithm	original algorithm	NewTop protocol algorithm			**Lamport's algorithm**

47	The _____ was proposed by Chang and Maxemchuck for various variants of it were given by the Pinwheel and RMP algorithms.	Lamport's algorithm	original algorithm	Pass algorithm	NewTop protocol algorithm			**original algorithm**
48	When a _____ receives the token, it assigns a sequence number to all received but unsequenced messages.	parser	detector	sequencer	failure			**sequencer**
49	Moving sequencer algorithms gurantee_____.	partial ordering	unpartial ordering	timestamped	total ordering			**total ordering**
50	The unicast from the sender to the sequencer as well as the multicast from the sequencer are _____	batch processing	stack holding	fault tolerance.	timestamped.			**timestamped.**
51	When a system component fails in the midst of the _____, which is a non atomic operation that spans across time and across multiple links and nodes the behaviour of a multicast protocol must adhere to a well defined specification.	unicast operation	multiprocessing	multi tasking	multicast operation			**multicast operation**
52	In the regular flavour there are no conditions on the messages delivered to_____.	multiprocessors	multi computers	faulty processors	single processors			**faulty processors**
53	The_____states that once the multicast is initiated by a correct process, it will go to completion.	invalidating	validity property	validating	processing			**validity property**
54	It is time to remember the folklore result that any protocol or implementation that deals with fault tolerance incurs a greater cost than what it would in a _____environment.	failure free	failure occurence	fault tolerance.	fault detection			**failure free**
55	_____ in delivering a multicast message can also be viewed as a fault.	timing delay	Excessive delay	signal delay	proceesing delay			**Excessive delay**
56	Several applications can interface directly with the network layer and the_____related layers to exploit the physical connectivity and the physical topology for group communication.	higher hardware	higher software	lower hardware	lower software			**lower hardware**
57	Network nodes are assumed to run the _____ routing algorithm which was used in the internet until 1983	magnitude	vector	distance and magnitude vector	distance vector			**distance vector**
58	_____ and interactive applications have given rise to the need for a minimum Steiner tree that also satisfies delay constraints on the transmission.	networks	processors	links	Multimedia networks			**Multimedia networks**
59	At the core of distributed computing is the communication by message passing among the processes participating in the_____.	software	hardware	middleware	application			**application**
60	Maintaining _____ in the presence of faults is necessary in real world systems.	links	terminals	communication	layers			**communication**

Termination detection

Introduction

In distributed processing systems, a problem is typically solved in a distributed manner with the cooperation of a number of processes. In such an environment, inferring if a distributed computation has ended is essential so that the results produced by the computation can be used. Also, in some applications, the problem to be solved is divided into many subproblems, and the execution of a subproblem cannot begin until the execution of the previous subproblem is complete. Hence, it is necessary to determine when the execution of a particular subproblem has ended so that the execution of the next subproblem may begin. Therefore, a fundamental problem in distributed systems is to determine if a distributed computation has terminated.

The detection of the termination of a distributed computation is non-trivial since no process has complete knowledge of the global state, and global time does not exist. A distributed computation is considered to be globally terminated if every process is locally terminated and there is no message in transit between any processes. A "locally terminated" state is a state in which a process has finished its computation and will not restart any action unless it receives a message. In the termination detection problem, a particular process (or all of the processes) must infer when the underlying computation has terminated.

When we are interested in inferring when the underlying computation has ended, a termination detection algorithm is used for this purpose. In such situations, there are two distributed computations taking place in the distributed system, namely, the *underlying computation* and the *termination detection algorithm*. Messages used in the underlying computation are called *basic* messages, and messages used for the purpose of termination detection (by a termination detection algorithm) are called *control* messages.

A termination detection (TD) algorithm must ensure the following:

1. Execution of a TD algorithm cannot indefinitely delay the underlying computation; that is, execution of the termination detection algorithm must not freeze the underlying computation.
2. The termination detection algorithm must not require addition of new communication channels between processes.

3.1 System model of a distributed computation

A distributed computation consists of a fixed set of processes that communicate solely by message passing. All messages are received correctly after an arbitrary but finite delay. Communication is *asynchronous*, i.e., a process never waits for the receiver to be ready before sending a message. Messages sent over the same communication channel may not obey the FIFO ordering.

A distributed computation has the following characteristics:

1. At any given time during execution of the distributed computation, a process can be in only one of the two states: *active*, where it is doing local computation and *idle*, where the process has (temporarily) finished the execution of its local computation and will be reactivated only on the receipt of a message from another process. The active and idle states are also called the *busy* and *passive* states, respectively.

2. An active process can become idle at any time. This corresponds to the situation where the process has completed its local computation and has processed all received messages.
3. An idle process can become active only on the receipt of a message from another process. Thus, an idle process cannot spontaneously become active (except when the distributed computation begins execution).
4. Only active processes can send messages. (Since we are not concerned with the initialization problem, we assume that all processes are initially idle and a message arrives from outside the system to start the computation.)
5. A message can be received by a process when the process is in either of the two states, i.e., *active* or *idle*. On the receipt of a message, an *idle* process becomes *active*.
6. The sending of a message and the receipt of a message occur as atomic actions.
We restrict our discussion to executions in which every process eventually becomes idle, although this property is in general undecidable. If a termination detection algorithm is applied to a distributed computation in which some processes remain in their active states forever, the TD algorithm itself will not terminate.

Definition of termination detection
Let $p_i(t)$ denote the state (active or idle) of process p_i at instant t and $c_{i,j}(t)$ denote the number of messages in transit in the channel at instant t from process p_i to process p_j . A distributed computation is said to be terminated at time instant t_0 iff:

$$(\forall i :: p_i(t_0) = idle) \wedge (\forall i, j :: c_{i,j}(t_0) = 0).$$

3.2 Termination detection using distributed snapshots
The algorithm uses the fact that a consistent snapshot of a distributed system captures stable properties. Termination of a distributed computation is a stable property. Thus, if a consistent snapshot of a distributed computation is taken after the distributed computation has terminated, the snapshot will capture the termination of the computation.
The algorithm assumes that there is a logical bidirectional communication channel between every pair of processes. Communication channels are reliable but non-FIFO. Message delay is arbitrary but finite.

3.2.1 Informal description
The main idea behind the algorithm is as follows: when a computation terminates, there must exist a unique process which became idle last. When a process goes from active to idle, it issues a request to all other processes to take a local snapshot, and also requests itself to take a local snapshot. When a process receives the request, if it agrees that the requester became idle before itself, it grants the request by taking a local snapshot for the request. A request is said to be *successful* if all processes have taken a local snapshot for it. The requester or any external agent may collect all the local snapshots of a request. If a request is successful, a global snapshot of the request can thus be obtained and the recorded state will indicate termination of the computation, viz., in the recorded snapshot, all the processes are idle and there is no message in transit to any of the processes.

3.2.2 Formal description

The algorithm needs logical time to order the requests. Each process *i* maintains an *logical clock* denoted by x, which is initialized to zero at the start of the computation. A process increments its x by one each time it becomes idle.

A basic message sent by a process at its logical time x is of the form *B*(x). A control message that requests processes to take local snapshot issued by process i at its logical time x is of the form *R*(x, i). Each process synchronizes its logical clock *x* loosely with the logical clocks *x*'s on other processes in such a way that it is the maximum of clock values ever received or sent in messages. Besides logical clock *x*, a process maintains a variable k such that when the process is idle, (x,k) is the maximum of the values (x, k) on all messages *R(x, k)* ever received or sent by the process. Logical time is compared as follows: (x, k) > (x', k') iff (x > x') or ((x = x') and (k > k')), i.e., a tie between *x* and x' is broken by the process identification numbers *k* and k'.

The algorithm is defined by the following four rules. We use guarded statements to express the conditions and actions. Each process *i* applies one of the rules whenever it is applicable.

R1: When process i is active, it may send a basic message to process j at any time by doing

send a $B(x)$ *to* j.

R2: Upon receiving a $B(x')$, process i does

let $x := x' + 1$;
if (i is *idle*) $\rightarrow$ go *active*.

R3: When process i goes *idle*, it does

let $x := x + 1$;
let $k := i$;
send message $R(x, k)$ to all other processes;
take a local snapshot for the request by $R(x, k)$.

R4: Upon receiving message $R(x', k')$, process i does

$[((x', k') > (x, k)) \wedge (i \text{ is } idle) \rightarrow \text{let}(x, k) := (x', k')$;
take a local snapshot for the request by $R(x', k')$;
□
$((x', k') \leq (x, k)) \wedge (i \text{ is } idle) \rightarrow$ do nothing;
□
$(i \text{ is } active) \rightarrow$ let $x := max(x', x)]$.

3.3 Termination detection in a faulty distributed system

An algorithm is presented that detects termination in distributed systems in which processes fail in a fail-stop manner. The algorithm is based on the weight-throwing method. In such a distributed system, a computation is said to be terminated if and only if each healthy process is idle and there is no basic message in transit whose destination is a healthy process. This is independent of faulty processes and undeliverable messages (i.e., whose destination is a

faulty process). Based on the weight-throwing scheme, a scheme called flow detecting scheme is developed by Tseng to derive a fault-tolerant termination detection algorithm.

Assumptions

Let S = P1, P2, , Pn be the set of processes in the distributed computation. Cij represents the bidirectional channel between Pi and Pj . The communication network is asynchronous. Communications channels are reliable, but they are non-FIFO. At any time, an arbitrary number of processes may fail. However, the network remains connected in the presence of faults. The fail-stop model implies that a failed process stops all activities and cannot rejoin the computation in the current session. Detection of faults takes a finite amount of time.

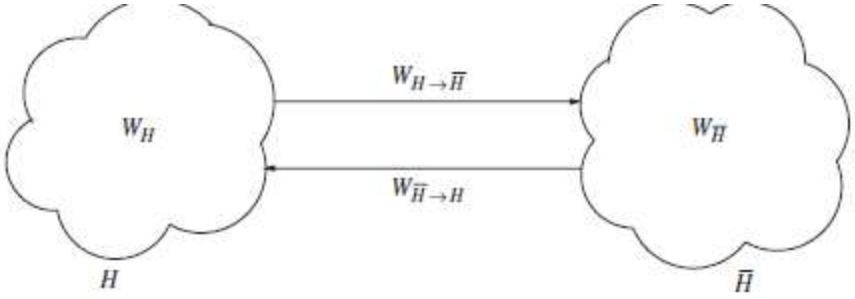
3.3.1 Flow detecting scheme

Weights may be lost because a process holding a non-zero weight may crash or a message destined to a crashed process is carrying a weight. Therefore, due to faulty processes and undeliverable messages carrying weights, it may not be possible for the leader to accumulate the total weight of 1 to declare termination. In the case of a process crash, the lost weight must be calculated. To solve this problem, the concept of flow invariant is used.

The concept of flow invariant

Define $H \subseteq S$ as the set of all healthy processes. Define *subsystem* H to be part of the system containing all processes in H and communication channels connecting two processes in H. According to the concept of flow invariant, the weight change of the subsystem during time interval I, during which the system is doing computation, is equal to (weights flowing into H during I) − (weights flowing out of H during I). To implement this concept, a variable called neti is assigned to each process Pi belonging to H. This variable records the total weight flowing into and out of the subsystem H. Initially, $\forall i\ net_i = 0$. The following flow-detecting rules are defined:

Rule 1: Whenever a process Pi which belongs to H receives a message with weight x from another process Pj which does not belong to H, x is added to net_i.



Rule 2: Whenever a process P_i which belongs to H sends a message with weight x to a process P_j which does not belong to H, x is subtracted from net_i.

Let W_H be the sum of the weights of all processes in H and all in-transit messages transmitted between processes in H:

$$W_H = \sum_{P_i \in H} (net_i + 1/n),$$

where $1/n$ is the initial weight held by each process P_i.

Let $\overline{H} = S - H$ be the set of faulty processes. The distribution of weights is divided into four parts:

W_H: weights of processes in H.
$W_{\overline{H}}$: weights of processes in $\overline{H}$.
$W_{H\to\overline{H}}$: weights held by in-transit messages from H to $\overline{H}$.
$W_{\overline{H}\to H}$: weights held by in-transit messages from $\overline{H}$ to H.

This is shown in Figure 7.11. $W_{\overline{H}}$ and $W_{H\to\overline{H}}$ are lost and cannot be used in the termination detection.

3.4. Distributed mutual exclusion algorithms

Mutual exclusion is a fundamental problem in distributed computing systems. Mutual exclusion ensures that concurrent access of processes to a shared resource or data is serialized, that is, executed in a mutually exclusive manner. Mutual exclusion in a distributed system states that only one process is allowed to execute the critical section (CS) at any given time. In a distributed system, shared variables (semaphores) or a local kernel cannot be used to implement mutual exclusion. Message passing is the sole means for implementing distributed mutual exclusion. The decision as to which process is allowed access to the CS next is arrived at by message passing, in which each process learns about the state of all other processes in some consistent way. The design of distributed mutual exclusion algorithms is complex because these algorithms have to deal with unpredictable message delays and incomplete knowledge of the system state. There are three basic approaches for implementing distributed mutual exclusion:

1. Token-based approach.
2. Non-token-based approach.
3. Quorum-based approach.

In the token-based approach, a unique token (also known as the PRIVILEGE message) is shared among the sites. A site is allowed to enter its CS if it possesses the token and it continues to hold the token until the execution of the CS is over. Mutual exclusion is ensured because the token is unique. The algorithms based on this approach essentially differ in the way a site carries out the search for the token. In the non-token-based approach, two or more successive rounds of messages are exchanged among the sites to determine which site will enter the CS next. A site enters the critical section (CS) when an assertion, defined on its local variables, becomes true. Mutual exclusion is enforced because the assertion becomes true only at one site at any given time. In the quorum-based approach, each site requests permission to execute the CS from a subset of sites (called a quorum). The quorums are formed in such a way that when two sites concurrently request access to the CS, at least one site receives both the requests and this site is responsible to make sure that only one request executes the CS at any time. In this chapter, we describe several distributed mutual exclusion algorithms and compare their features and performance. We discuss relationship among various mutual exclusion algorithms and examine trade-offs among them.

3.5 Lamport's algorithm

Lamport developed a distributed mutual exclusion algorithm as an illustration of his clock synchronization scheme. The algorithm is fair in the sense that a request for CS are executed in the order of their timestamps and time is determined by logical clocks. When a site processes a request for the CS, it updates its local clock and assigns the request a timestamp. The algorithm executes CS requests in the increasing order of timestamps. Every site Si keeps a queue, request_queuei, which contains mutual exclusion requests ordered by their timestamps. (Note that this queue is different from the queue that contains local requests for CS execution awaiting their turn.) This algorithm requires communication channels to deliver

messages in FIFO order

Requesting the critical section

- When a site S_i wants to enter the CS, it broadcasts a REQUEST(ts_i, i) message to all other sites and places the request on $request\_queue_i$. ((ts_i, i) denotes the timestamp of the request.)
- When a site S_j receives the REQUEST(ts_i, i) message from site S_i, it places site S_i's request on $request\_queue_j$ and returns a timestamped REPLY message to S_i.

Executing the critical section

Site S_i enters the CS when the following two conditions hold:

L1: S_i has received a message with timestamp larger than (ts_i, i) from all other sites.

L2: S_i's request is at the top of $request\_queue_i$.

Releasing the critical section

- Site S_i, upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped RELEASE message to all other sites.
- When a site S_j receives a RELEASE message from site S_i, it removes S_i's request from its request queue.

Algorithm 9.1 Lamport's algorithm.

When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS. Clearly, when a site receives a REQUEST, REPLY, or RELEASE message, it updates its clock using the timestamp in the message.

Correctness

Theorem 1 *Lamport's algorithm achieves mutual exclusion.*

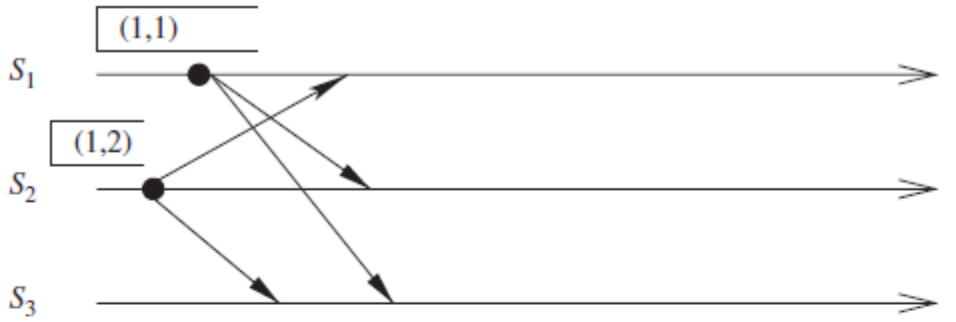
Proof is by contradiction. Suppose two sites Si and Sj are executing the CS concurrently. For this to happen conditions L1 and L2 must hold at both the sites *concurrently*. This implies that at some instant in time, say t, both Si and Sj have their own requests at the top of their request_queues and condition L1 holds at them. Without loss of generality, assume that Si's request has smaller timestamp than the request of Sj . From condition L1 and FIFO property of the communication channels, it is clear that at instant t the request of Si must be present in request_queuej when Sj was executing its CS. This implies that Sj's own request is at the top of its own request_queue when a smaller timestamp request, Si's request, is present in the request_queuej – a contradiction! Hence, Lamport's algorithm achieves mutual exclusion.

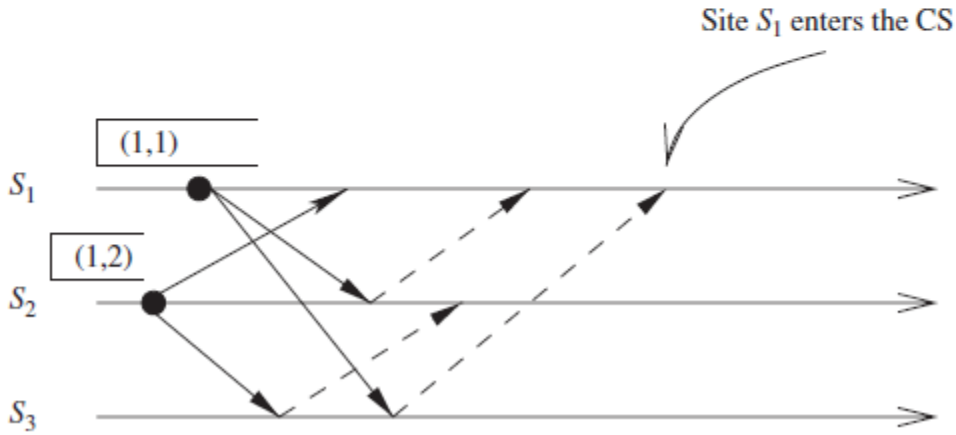
Theorem 2 *Lamport's algorithm is fair.*

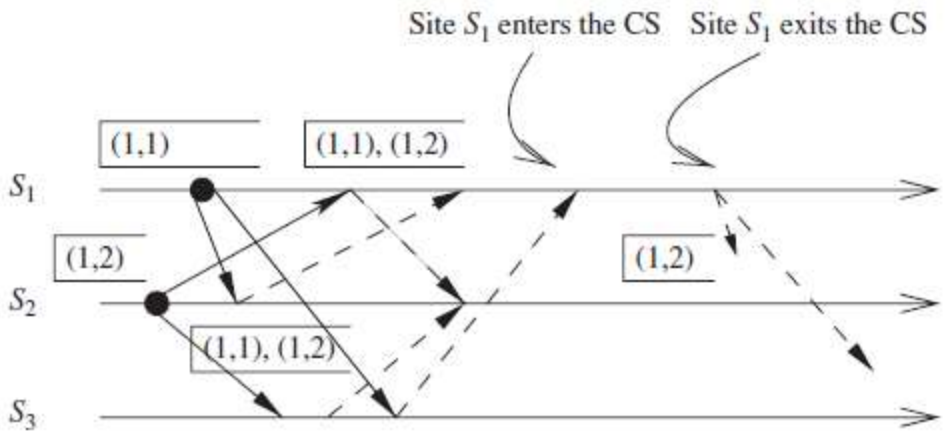
Proof A distributed mutual exclusion algorithm is fair if the requests for CS are executed in the order of their timestamps. The proof is by contradiction. Suppose a site Si's request has a smaller timestamp than the request of another site Sj and Sj is able to execute the CS before

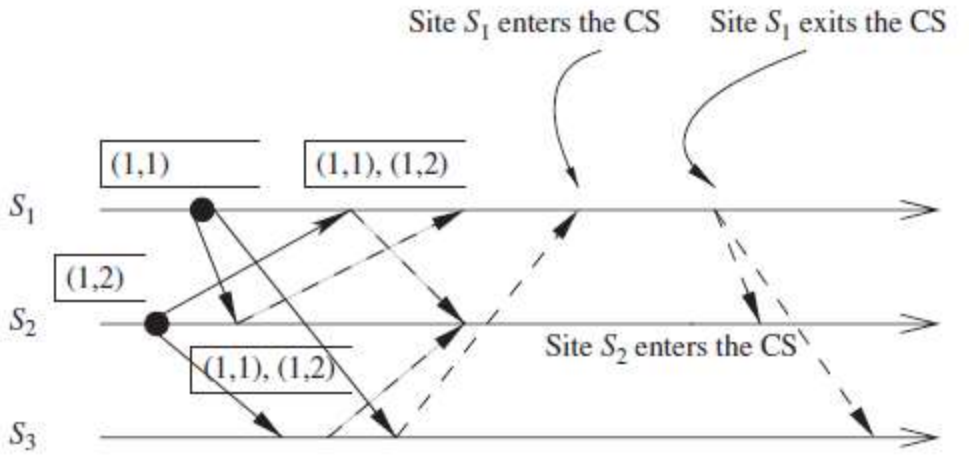
Si. For Sj to execute the CS, it has to satisfy the conditions L1 and L2. This implies that at some instant in time Sj has its own request at the top of its queue and it has also received a message with timestamp larger than the timestamp of its request from all other sites. But request_queue at a site is ordered by timestamp, and according to our assumption Si has lower timestamp. So Si's request must be placed ahead of the Sj's request in the request_queuej . This is a contradiction. Hence Lamport's algorithm is a fair mutual exclusion algorithm.

Example In Figure, sites S1 and S2 are making requests for the CS and send out REQUEST messages to other sites. The timestamps of the requests are (1,1) and (1,2), respectively. In Figure 9.4, both the sites S1 and S2 have received REPLY messages from all other sites. S1 has its request at the top of its request_queue but site S2 does not have its request at the top of its request_queue. Consequently, site S1 enters the CS. In Figure, S1 exits and sends RELEASE mesages to all other sites. In Figure, site S2 has received REPLY from all other sites and also received a RELEASE message









from site S1. Site S2 updates its request_queue and its request is now at the top of its request_queue. Consequently, it enters the CS next.

Performance

For each CS execution, Lamport's algorithm requires (N −1)REQUEST messages, (N −1) REPLY messages, and (N −1) RELEASE messages. Thus, Lamport's algorithm requires 3_N −1_ messages per CS invocation. The synchronization delay in the algorithm is T.

An optimization

In Lamport's algorithm, REPLY messages can be omitted in certain situations. For example, if site Sj receives a REQUEST message from site Si after it has sent its own REQUEST message with a timestamp higher than the timestamp of site Si's request, then site Sj need not send a REPLY message to site Si. This is because when site Si receives site Sj's request with a timestamp higher than its own, it can conclude that site Sj does not have any smaller timestamp request which is still pending (because communication channels preserves FIFO ordering). With this optimization, Lamport's algorithm requires between 3(N −1)and 2(N −1) messages per CS execution.

3.6 Token-based algorithms

In token-based algorithms, a unique token is shared among the sites. A site is allowed to enter its CS if it possesses the token. A site holding the token can enter its CS repeatedly until it sends the token to some other site. Depending upon the way a site carries out the search for the token, there are numerous token-based algorithms. Next, we discuss two token-based mutual exclusion algorithms.

Before we start with the discussion of token-based algorithms, two comments are in order. First, token-based algorithms use sequence numbers instead of timestamps. Every request for the token contains a sequence number and the sequence numbers of sites advance independently. A site increments its sequence number counter every time it makes a request for the token. (A primary function of the sequence numbers is to distinguish between old and current requests.) Second, the correctness proof of token-based algorithms, that they enforce mutual exclusion, is trivial because an algorithm guarantees mutual exclusion so long as a site holds the token during the execution of the CS. Instead, the issues of freedom from starvation, freedom from deadlock, and detection of the token loss and its regeneration become more prominent

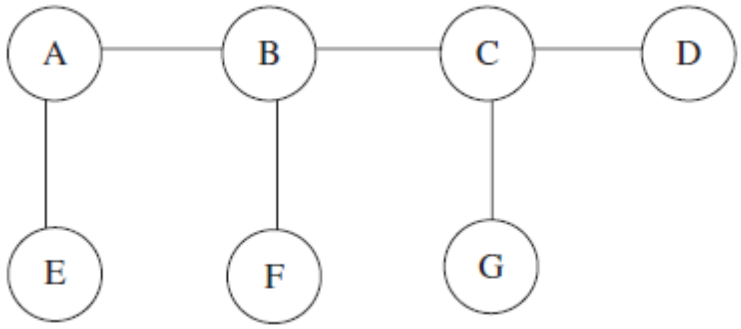
3.7 Raymond's tree-based algorithm

Raymond's tree-based mutual exclusion algorithm uses a spanning tree of the computer network to reduce the number of messages exchanged per critical section execution. The algorithm exchanges only O(log N) messages under light load, and approximately four messages under heavy load to execute the CS, where N is the number of nodes in the network. The algorithm assumes that the underlying network guarantees message delivery. The time or order of message arrival cannot be predicted. All nodes of the network are completely reliable. (Only for the initial part of the discussion, i.e., until node failure is discussed.) If the network is viewed as a graph, where the nodes in the network are the vertices of the graph, and the links between nodes are the edges of the graph, a spanning tree of a network of N nodes will be a tree that contains all N nodes. A minimal spanning tree is one such tree with minimum cost. Typically, this cost function is based on the network link characteristics. The algorithm operates on a minimal spanning tree of the network topology or logical structure imposed on the network.

The algorithm considers the network nodes to be arranged in an unrooted tree structure as shown in Figure. Messages between nodes traverse along the undirected edges of the tree in the Figure. The tree is also a spanning tree of the seven nodes A, B, C, D, E, F, and G. It also turns out to be a minimal spanning tree because it is the only spanning tree of these seven nodes. A node needs to hold information about and communicate only to its immediate-neighboring nodes. In Figure 9.17, for example, node C holds information about and communicates only to nodes B, D, and G; it does not need to know about the other nodes A, E, and F for the operation of the algorithm.

Similar to the concept of tokens used in token-based algorithms, this algorithm uses a concept of privilege to signify which node has the privilege to enter the critical section. Only one node can be in possession of the privilege (called the privileged node) at any time, except

when the privilege is in transit



from one node to another in the form of a PRIVILEGE message. When there are no nodes requesting for the privilege, it remains in possession of the node that last used it.

3.7.1 The HOLDER variables

Each node maintains a HOLDER variable that provides information about the placement of the privilege in relation to the node itself. A node stores in its HOLDER variable the identity of a node that it thinks has the privilege or leads to the node having the privilege. The HOLDER variables of all the nodes maintain directed paths from each node to the node in the possession of the privilege.

For two nodes X and Y, if HOLDERX= Y, we could redraw the undirected edge between the nodes X and Y as a directed edge from X to Y. Thus, for instance, if node G holds the privilege, Figure can be redrawn with logically directed edges as shown in Figure. The shaded node represents the privileged node. The following will be the values of the HOLDER variables of various nodes:

$HOLDER_A = B$(Since the privilege is located in a sub-tree of A denoted by B.)

Proceeding with similar reasoning, we have

$$HOLDER_B = C,$$
$$HOLDER_C = G,$$
$$HOLDER_D = C,$$
$$HOLDER_E = A,$$
$$HOLDER_F = B,$$
$$HOLDER_G = self.$$

Now suppose that node B, which does not hold the privilege, wants to execute the critical section. Then B sends a REQUEST message to $HOLDER_B$, i.e., C, which in turn forwards the REQUEST message to $HOLDER_C$, i.e., G. So a series of REQUEST messages flow between the node making the request for the privilege and the node having the privilege.

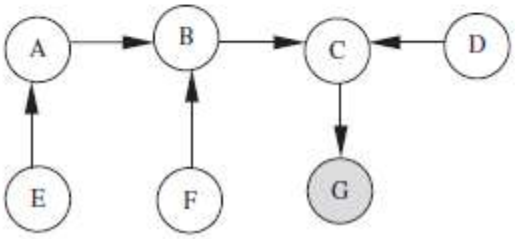
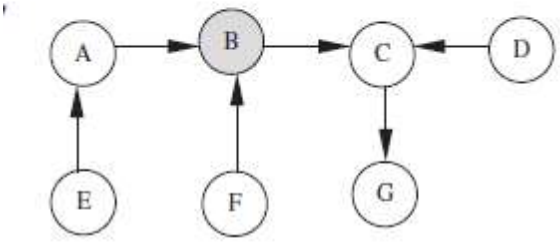


Table 9.1 Variables used in the algorithm.

Variable name	Possible values	Comments
HOLDER	"self" or the identity of one of the immediate neighbors.	Indicates the location of the privileged node in relation to the current node.
USING	True or false.	Indicates if the current node is executing the critical section.
REQUEST_Q	A FIFO queue that could contain "self" or the identities of immediate neighbors as elements.	The REQUEST_Q of a node consists of the identities of those immediate neighbors that have requested for privilege but have not yet been sent the privilege.
ASKED	True or false.	Indicates if node has sent a request for the privilege.



The privileged node G, if it no longer needs the privilege, sends the PRIVILEGE message to its neighbor C, which made a request for the privilege, and resets HOLDERG to C. Node C, in turn, forwards the PRIVILEGE to node B, since it had requested the privilege on behalf of B. Node C also resets HOLDERC to B. The tree in Figure will now look as shown in Figure. Thus, at any stage, except when the PRIVILEGE message is in transit, the HOLDER variables collectively make sure that directed paths are maintained from each of the $N - 1$ nodes to the privileged node in the network.

3.7.2 The operation of the algorithm

Data structures

Each node maintains variables that are defined in Table 9.1. The value "self" is placed in REQUEST_Q if the node makes a request for the privilege for its own use. The maximum size of REQUEST_Q of a node is the number of immediate neighbors +1 (for "self "). ASKED prevents the sending of duplicate requests for privilege, and also makes sure that the REQUEST_Qs of the various nodes do not contain any duplicate elements.

3.7.3 Description of the algorithm

The algorithm consists of the following parts:

• ASSIGN_PRIVILEGE;

• MAKE_REQUEST;

• events;
• message overtaking.

ASSIGN_PRIVILEGE

This is a routine to effect the sending of a PRIVILEGE message. A privileged node will send a PRIVILEGE message if:
• it holds the privilege but is not using it;
• its REQUEST_Q is not empty; and
• the element at the head of its REQUEST_Q is not "self." That is, the oldest request for privilege must have come from another node.

A situation where "self" is at the head of REQUEST_Q may occur immediately after a node receives a PRIVILEGE message. The node will enter into the critical section after removing "self" from the head of REQUEST_Q. If the i.d. of another node is at the head of REQUEST_Q, then it is removed from the queue and a PRIVILEGE message is sent to that node. Also, the variable ASKED is set to false since the currently privileged node will not have sent a request to the node (called HOLDER-to-be) that is about to receive the PRIVILEGE message.

MAKE_REQUEST

This is a routine to effect the sending of a REQUEST message. An unprivileged node will send a REQUEST message if:
• it does not hold the privilege;
• its REQUEST_Q is not empty, i.e., it requires the privilege for itself, or on behalf of one of its immediate neighboring nodes; and
• it has not sent a REQUEST message already.

The variable ASKED is set to true to reflect the sending of the REQUEST message. The MAKE_REQUEST routine makes no change to any other variables. The variable ASKED will be true at a node when it has sent REQUEST message to an immediate neighbor and has not received a response. The variable will be false otherwise. A node does not send any REQUEST messages, if ASKED is true at that node. Thus the variable ASKED makes sure that unnecessary REQUEST messages are not sent from the unprivileged node, and consequently ensures that the REQUEST_Q of an immediate neighbor does not contain duplicate entries of a neighboring node. This makes the REQUEST_Q of any node bounded, even when operating under heavy load.

Table 9.2 Events in the algorithms.

Event	Algorithm functionality
A node wishes to execute critical section.	Enqueue (REQUEST_Q, Self); ASSIGN_PRIVILEGE; MAKE_REQUEST
A node receives a REQUEST message from one of its immediate neighbors X.	Enqueue(REQUEST_Q, X); ASSIGN_PRIVILEGE; MAKE_REQUEST
A node receives a PRIVILEGE message.	HOLDER := self; ASSIGN_PRIVILEGE; MAKE_REQUEST
A node exits the critical section.	USING := false; ASSIGN_PRIVILEGE; MAKE_REQUEST

Events

The four events that constitute the algorithm are shown in Table.

• **A node wishes critical section entry** If it is the privileged node, the node could enter the critical section using the ASSIGN_PRIVILEGE routine. If not, the node could send a REQUEST message using the MAKE_REQUEST routine in order to get the privilege.

• **A node receives a REQUEST message from one of its immediate neighbors** If this node is the current HOLDER, it may send the PRIVILEGE to a requesting node using the ASSIGN_PRIVILEGE routine. If not, it could forward the request using the MAKE_REQUEST routine.

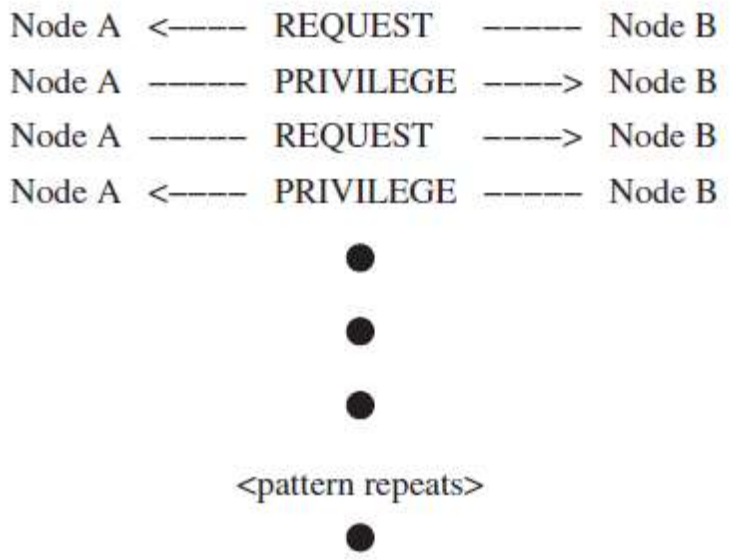
• **A node receives a PRIVILEGE message** The ASSIGN_PRIVILEGE routine could result in the execution of the critical section at the node, or may forward the privilege to another node. After the privilege is forwarded, the MAKE_REQUEST routine could send a REQUEST message to reacquire the privilege, for a pending request at this node.

• **A node exits the critical section** On exit from the critical section, this node may pass the privilege on to a requesting node using the ASSIGN_PRIVILEGE routine. It may then use the MAKE_REQUEST routine to get back the privilege, for a pending request at this node.

Message overtaking

This algorithm does away with the use of sequence numbers because of its inherent operations and by the acyclic structure it employs. Figure shows the logical pattern of message flow between any two neighboring nodes (nodes A and B here).

If any message overtaking occurs between nodes A and B, it can occur when a PRIVILEGE message is sent from node A to node B, which is then very closely followed by a REQUEST message from node A to node B. In other words, node A sends the privilege and immediately wants it back. Such



message overtaking as described above will not affect the operation of the algorithm. If node B receives the REQUEST message from node A before receiving the PRIVILEGE message from node A, A's request will be queued in REQUEST_QB. Since B is not a privileged node, it will not be able to send a privilege to node A in reply. When node B receives the PRIVILEGE message from A after receiving the REQUEST message, it could enter the critical section or could send a PRIVILEGE message to an immediate neighbor at the head of REQUEST_QB, which need not be node A. So message overtaking does not affect the algorithm.

3.7.4 Correctness

The algorithm provides the following guarantees:

- mutual exclusion is guaranteed;
- deadlock is impossible;
- starvation is impossible.

Mutual exclusion is guaranteed

The algorithm ensures that, at any instant of time, no more than one node holds the privilege, which is a necessity for mutual exclusion. Whenever a node receives a PRIVILEGE message, it becomes privileged. Similarly, whenever a node sends a PRIVILEGE message, it becomes unprivileged. Between the instants one node becomes unprivileged and another node becomes privileged, there is no privileged node. Thus, there is at most one privileged node at any point of time in the network.

Deadlock is impossible

When the critical section is free, and one or more nodes want to enter the critical section but are not able to do so, a deadlock may occur. This could happen due to any of the following scenarios:

1. The privilege cannot be transferred to a node because no node holds the privilege.
2. The node in possession of the privilege is unaware that there are other nodes requiring the privilege.
3. The PRIVILEGE message does not reach the requesting unprivileged node. None of the above three scenarios can occur in this algorithm, thus guarding against deadlocks. Scenario 1 can never occur in this algorithm because we have assumed that nodes do not fail and messages are not lost. There can never be a situation where REQUEST messages do not arrive at the privileged node. The logical pattern established using HOLDER variables ensures that a node that needs the privilege sends a REQUEST message either to a node holding the privilege or to a node that has a path to a node holding the privilege. Thus scenario 2 can never occur in this algorithm. The series of REQUEST messages are enqueued in the REQUEST_Qs of various nodes such that the REQUEST_Qs of those nodes collectively provide a logical path for the transfer of the PRIVILEGE message from the privileged node to the requesting unprivileged nodes. So scenario 3 can never occur in this algorithm.

Starvation is impossible

When node A holds the privilege, and node B requests the privilege, the identity of B or the i.d.s of the proxy nodes for node B will be present in the REQUEST_Qs of various nodes in the path connecting the requesting node to the currently privileged node. So, depending upon the position of the i.d. of node B in those REQUEST_Qs, node B will sooner or later receive the privilege. Thus once node B's REQUEST message reaches the privileged node A, node B is sure to receive the privilege.

To better illustrate, let us consider Figure. Node B is the current holder of the privilege. Suppose that node C is already at the head of REQUEST_QB. Assume that the REQUEST_Qs of all other nodes are empty. Now if node E wants to enter the critical section, it will send a REQUEST message to its immediate neighbor, node A. We will show that node E does not starve. Assume that B is executing the critical section by the time E's REQUEST is propagated to node B. At this instance, the REQUEST_Qs of E, A, and B will be as follows:

$REQUEST\_Q_E$ = self
$REQUEST\_Q_A$ = E
$REQUEST\_Q_B$ = C,A.

When node B exits the critical section, it removes the node at the head of REQUEST_QB, i.e., node C, and send the privilege to node C. Node B will then send a REQUEST to node C on behalf of node A, which requested privilege on behalf of node E. After node C receives the privilege and completes executing the critical section, the REQUEST_Qs of nodes C, B, A, and E will look as follows:

$REQUEST\_Q_C$ = B
$REQUEST\_Q_B$ = A
$REQUEST\_Q_A$= E_
$REQUEST\_Q_E$ = self_

Now, the next node to receive the privilege will be node E, a fact that is represented by the logical path "BAE" that the REQUEST_Qs of nodes C, B, and A form. Since node B had

requested privilege on behalf of node A, and node A on behalf of node E, the PRIVILEGE ultimately gets propagated to node E. Thus, a node never starves.

3.7.5 Cost and performance analysis

The algorithm exhibits the following worst-case cost: (2 * longest path length of the tree) messages per critical section entry. This happens when the privilege is to be passed between nodes at either end of the longest path of the minimal spanning tree. Thus the worst possible network topology for this algorithm will be one where all nodes are arranged in a straight line. In a straight line the longest path length will be N – 1, and thus the algorithm will exchange 2 * (N – 1) messages per CS execution. However, if all nodes generate equal number of REQUEST messages for the privilege, the average number of messages needed per critical section entry will be approximately 2N/3 because the average distance between a requesting node and a privileged node is (N +1)/3. The best topology for the algorithm is the radiating star topology. The worst-case cost of this algorithm for this topology is O(logK−1N). Even among radiating star topologies, trees with higher fan-outs are preferred. The longest path length of such trees is typically O(log N). Thus, on average, this algorithm involves the exchange of O(log N) messages per critical section execution. When under heavy load, the algorithm exhibits an interesting property: "as the number of nodes requesting the privilege increases, the number of messages exchanged per critical section entry decreases." In fact, it requires the exchange of only four messages per CS execution as explained below. When all nodes are sending privilege requests, PRIVILEGE messages travel along all N – 1 edges of the minimal spanning tree exactly twice to give the privilege to all N nodes. Each of these PRIVILEGE messages travel in response to a REQUEST message. Thus, a total of 4 * (N – 1) messages travel across the minimal spanning tree. Hence, the total number of messages exchanged per critical section execution is 4(N−1)/N, which is approximately 4.

3.7.6 Algorithm initialization

Algorithm initialization begins with one node being chosen as the privileged node. This node then sends INITIALIZE messages to its immediate neighbors. On receiving the INITIALIZE message, a node sets its HOLDER variable to the node that sent the INITIALIZE message, and send INITIALIZE messages to its own immediate neighbors. Once INITIALIZE message is received, anode can start making privilege requests even if the entire tree is not initialized. The initialization of the following variables is the same at all nodes:

USING := false
ASKED := false
REQUEST_Q := empty

3.7.7 Node failures and recovery

If a node fails, lost information can be reconstructed on restart. Once a node restarts, it enters into a recovery phase and selects a delay period for the recovery phase in order to get back all the lost information. It sends RESTART messages to its immediate neighbors and waits for ADVISE messages. During the recovery phase, the node can still receive REQUEST and PRIVILEGE messages; it acts as any normal node would act in response to those messages

except that ASSIGN_PRIVILEGE and MAKE_REQUEST routines are not executed. The ADVISE message that a recovering node A receives from each immediate neighbor B will contain information on the HOLDER, ASKED, and REQUEST_Q variables of B, from which A can reconstruct its own HOLDER, ASKED, and REQUEST_Q variables.

For example, if $HOLDER_B$ = A for all immediate neighbors B of node A, it means node A holds the privilege, and hence $HOLDER_A$ = self. Similar reasoning can be applied to determine value of $ASKED_A$ and the elements of $REQUEST\_Q_A$. $REQUEST\_Q_A$ can be reconstructed but the elements may not be in proper order. To ensure proper order, the ADVISE messages could provide real or logical timestamps for its REQUEST messages. USINGA can be set to false.

The recovering node's REQUEST_Q can have duplicates if it processes REQUEST messages sent currently and the ones it receives in the ADVISE messages. However, this does not affect the working of the algorithm as long as the REQUEST_Q is large enough to accommodate such situations. A node can also possibly fail when recovering from an earlier failure. In such a case, ASSIST messages related to the first recovery phase can be identified by making use of the delay chosen for recovery or unique identifiers, and those messages can be discarded.

Karpagam Academy of Higher Education
Department of Computer Applications
Subject Code/Name : Elective:17CAP505N/Distributed Computing

Objective Type Questions

	UNIT 3							
S.No	**Questions**	**OPTION 1**	**OPTION 2**	**OPTION 3**	**OPTION 4**	**OPTION 5**	**OPTION 6**	**Answer Key**
1	In _____ a problem is typically solved in a distributed manner with the cooperation of a number of processes.	distributed processing systems	DS	DC	OS			**distributed processing systems**
2	A fundamental problem in DS is to determine if a distributed computation has _____.	unterminated	terminated	processing	batching			**terminated**
3	A distributed computation is considered to be _____ if every process is locally terminated and there is no message in transits between any processes.	locally terminated	locally unterminated	globally terminated	globally unterminated			**globally terminated**
4	In the _____ problem a particular process must infer when the underlying computation has term.	termination avoidance	temination acceptance	termiantion	termination detection			**termination detection**
5	Messages used in the underlying computation are called _____.	static messages	dynamic messages	message creation	basic messages			**basic messages**
6	Messages are used for the purpose of termination detection are called _____.	static messages	dynamic messages	control messages	basic messages			**control messages**
7	The termination detection algorithm must not require addition of new communication channels between _____.	termination	processes	detection	avoidance			**processes**
8	A distributed computation consists of a fixed set of processes that communicate solely by _____.	message passing	message creations	message detection	message deletion			**message passing**
9	A Process never waits for the receiver to be ready before sending a _____.	packets	message	arrays	numbers			**message**
10	Messages sent over the same communication channel may not obey the _____.	LIFO ordering	stack	FIFO ordering	queue			**FIFO ordering**
11	The algorithm uses than fact that a consistent snapshot of a distributed system captures _____.	unstable properties	implicable	imposed	stable properties			**stable properties**
12	Termination of a_____ is a stable property, if a consistent snapshot of a DC is taken after the DC is terminated, the snapshot will capture the termination of the computation.	DS	Distributed methods	distributed computation	teriminology			**distributed computation**
13	The_____ assumes that there is a logical bidirectional communication channel between every pair of processes.	procedures	stacks	queues	algorithm			**algorithm**
14	Communication channels are reliable but non_FIFO, message delay is arbitrary but _____.	infinite	unlimited	finite	limited			**finite**
15	When a process goes from active to idle , it issues a request to all other processes to take a local snapshot and also requests itself to take the _____.	global snapshot	local snapshot	globally terminated	locally terminated			**local snapshot**

16	In the _____all the processes are idle and there is no message in transits to any of the processes.	recorded snapshot	snapshot	fully connected snapshot	fully terminated snapshot			**recorded snapshot**
17	The algorithm needs _____to order the requests.	physical time	logical time	allocating time	terminating time			**logical time**
18	In by weight throwing _____a process called controlling agent monitors the computation.	termination avoidance	termination	termination detection	termination communicated			**termination detection**
19	A _____ exists between each of the processes and the controlling agent and also between every pairs of processes.	pathway	stabled pathway	unstabled pathway	communication channel			**communication channel**
20	An algorithm is presented that detects termination in DS in which processes fail in a _____.	stop manner	fail manner	fail_stop manner	unstop manner			**fail_stop manner**
21	_____ may be lost because a process holding a non_zero weight may crash or a message designated toa crashed process is carrying a weight.	stacks	Weights	processes	terminals			**Weights**
22	In the case of a _____, the lost weight must be calculated.	process crash	system crash	terminals crash	task crash			**process crash**
23	To the problem of _____ the concept of flow invariant is used.	trashing	crashing	deleting	avoiding			**crashing**
24	The algorithm combines the weight throwing scheme the flow detecting scheme and a _____ recording scheme.	take a loop	lookahead	snapshot	weights			**snapshot**
25	The _____ takes snapshots and estimates remaining weight in the system.	processing	stop and wait	terminating process	leader process			**leader process**
26	The message complexity of the algorithm is _____.	O(M)	O(N)	O(M+kn+n)	O(M+kn)			**O(M+kn+n)**
27	A DC is terminated if every processes_____ and there is no message in transits between any processes.	globally termianted	locally terminated	locally started	globally startes			**locally terminated**
28	Determining if a DC has terminated is a_____ in DS.	fundamental problem	basci needs	solution for the problem	problem solving			**fundamental problem**
29	_____of the termination of a DC is a nontrivial task since no process has complete knowledge of the global state.	termination	Detection	avoidance	acceptance			**Detection**
30	_____ is a fundamental problem and it finds applications at several places in DS.	termination avoidance	termination acceptance	Termination detection	termination needs			**Termination detection**
31	_____ is a fundamental problem in DCS.	error detection	error correction	mutual avoidance	Mutual exclusion			**Mutual exclusion**
32	_____ ensures that concurrent access of processes to a shared resource or data is serialized that is executed in a mutually exclusive manner.	error detection	error correction	Mutual exclusion	mutual avoidance			**Mutual exclusion**
33	Mutual exclusion in a DS states that only one process is allowed to execute the critical section at any _____	limited time	given time	unlimited time	infinite time			**given time.**
34	In a DS shared variables or a _____cannot be used to implement mutual exclusion.	local kernel	global kernel	avoidance	detection			**local kernel**
35	_____ is the sole means for implementing distributed mutual exclusion.	message termination	Message passing	message looping	message acceptance			**Message passing**

36	The design of the _____ algorithms is complex because these algorithms have to deal with unpredictable message delays and incomplete knowledge of the system state.	accepted terminals	unaccepted terminals	distributed mutual exclusion	distributed systems			**distributed mutual exclusion**
37	In the token based approach a unique token is shared among the_____.	local terminals	stacks	processors	sites			**sites**
38	_____ is enforced because the assertion becomes true only at one site at any given time.	local kernel	global kernel	Mutual exclusion	detection and avoidance			**Mutual exclusion**
39	In the _____ each site requests permission to execute the CS from a subset of sites.	distibuted based approach	quorum based approach	mutually accepted approach	existing approach			**quorum based approach**
40	_____ are reliable but non_FIFO, message delay is arbitrary but finite.	Communication channels	terminals	system channels	links			**Communication channels**
41	A _____ to enter the CS requests all other or a subset of processes by sending REQUEST messages, and waits for appropriate replies before entering the CS.	process switching	process wishing	process termination	process accepting			**process wishing**
42	We assume that _____reliably deliver all messages sites do not crash and the network does not get partitioned.	swtiches	processes	channels	tasks			**channels**
43	_____ are used to decide the priority to critical section requests.	processors	kernels	terminals	Timestamps			**Timestamps**
44	The general rule followed is that the smaller the timestamp of a request the _____ its priority to execute the CS.	lower	very lower	very higher	higher			**higher**
45	_____ property is an essential property of a mutual exclusion algorithm.	unsafety	limited	unlimted	Safety			**Safety**
46	_____ should not endlessly wait for messages that will never arrive.	one	two	Two or more sites	more than five			**Two or more sites**
47	In the_____ the fairness property generally means that the CS execution requests are executed by in order of their arrival in the system.	algorithms	mutual exclusion algorithms	accepting algorithms	nonaccepting algorithms			**mutual exclusion algorithms**
48	The value of the _____ fluctuates statistically fro request to request and we generally consider the average value of such a metric.	performance metric	measures	task completed	predefined task			**performance metric**
49	For many _____ the performance metrics can be computed easily under low and heavy loads through a simple mathematical reasoning.	algorithms	accepting algorithms	mutual exclusion algorithms	nonaccepting algorithms			**mutual exclusion algorithms**
50	Lamport's developed a _____as an illustration of his clock synchronization scheme.	accepted terminals	nonaccepting mutual exclusion	terminals	distributed mutual exclusion algorithm			**distributed mutual exclusion algorithm**
51	The Richard Agrawala algorithm assumes that the communication channels are _____.	FIFO	LIFO model	stack processors	Array processors			**FIFO**
52	A process sends a _____ to a process to give its permission to that process.	SEND messages	REPLY message	RECEIVE messages	TRANSMIT messages			**REPLY message**
53	In _____ a unique token is shared among the sites.	system based algorithms	locally terminating	token based algorithms	globally terminating			**token based algorithms**

54	In Suzuki Kasami's algorithm, if a site that wants to enter the CS does not have the token it broadcasts a REQUEST message for the token to all _____.	local sites	global sites	different sites	other sites			**other sites**
55	Raymond's tree based _____ uses a spanning tree of the computer network to reduce the number of messages exchanged per critical section execution.	system based algorithms	locally terminating algorithms	mutual exclusion algorithm	globally terminating algorithms			**mutual exclusion algorithm**
56	_____ does away with the use of sequence numbers because of its inherent operations and by the acyclic structure it employs.	system based algorithms	Message overtaking algorithm	mutual exclusion algorithm	globally terminating algorithms			**Message overtaking algorithm**
57	When the _____ is free and one or more nodes want enter the critical situation but are not able to do so, a dead lock may occur.	critical section	local section	global section	same section			**critical section**
58	The privilege cannot be _____ to a node because no node holds the privilege.	untransferred	transferred	limited	unlimited			**transferred**
59	The_____ exhibits the worst case cost messages per critical section entry.	procedures	tasks	algorithm	functions			**algorithm**
60	The best topology for the algorithm is the radiating _____.	ring topology	bus topology	mesh topology	star topology			**star topology**

Deadlock detection in distributed systems

Deadlocks are a fundamental problem in distributed systems and deadlock detection in distributed systems has received considerable attention in the past. In distributed systems, a process may request resources in any order, which may not be known a priori, and a process can request a resource while holding others. If the allocation sequence of process resources is not controlled in such environments, deadlocks can occur. A deadlock can be defined as a condition where a set of processes request resources that are held by other processes in the set. Deadlocks can be dealt with using any one of the following three strategies: deadlock prevention, deadlock avoidance, and deadlock detection. Deadlock prevention is commonly achieved by either having a process acquire all the needed resources simultaneously before it begins execution or by pre-empting a process that holds the needed resource. In the deadlock avoidance approach to distributed systems, a resource is granted to a process if the resulting global system is safe. Deadlock detection requires an examination of the status of the process–resources interaction for the presence of a deadlock condition. To resolve the deadlock, we have to abort a deadlocked process.

4.1 System model

A distributed system consists of a set of processors that are connected by a communication network. The communication delay is finite but unpredictable. A distributed program is composed of a set of n asynchronous processes *P*1, *P*2, , *P*i, , *P*n that communicate by message passing over the communication network. Without loss of generality we assume that each process is running on a different processor. The processors do not share a common global memory and communicate solely by passing messages over the communication network. There is no physical global clock in the system to which processes have instantaneous access. The communication medium may deliver messages out of order, messages may be lost, garbled, or duplicated due to timeout and retransmission, processors may fail, and communication links may go down. The system can be modeled as a directed graph in which vertices represent the processes and edges represent unidirectional communication channels.

We make the following assumptions:

• The systems have only reusable resources.

• Processes are allowed to make only exclusive access to resources.

• There is only one copy of each resource.

A process can be in two states, *running* or *blocked*. In the running state (also called *active* state), a process has all the needed resources and is either executing or is ready for execution. In the blocked state, a process is waiting to acquire some resource.

4.2.1 Wait-for graph (WFG)

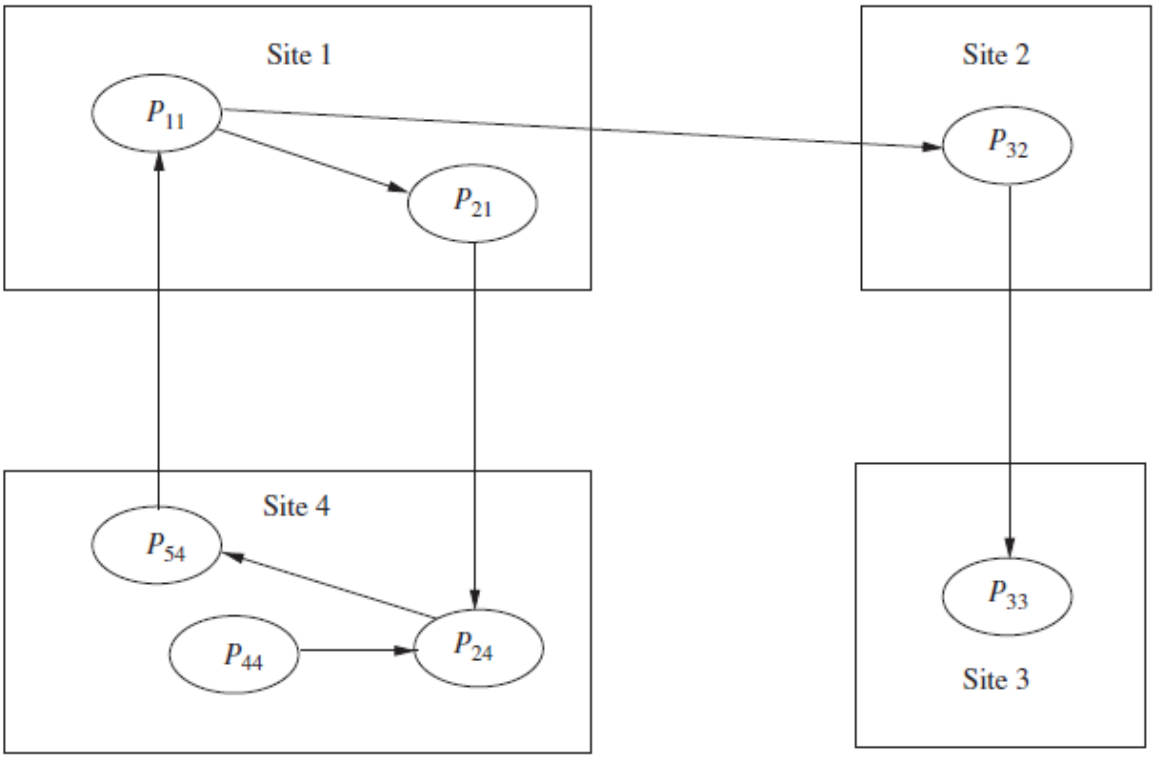
In distributed systems, the state of the system can be modeled by directed graph, called a *wait-for graph* (WFG). In a WFG, nodes are processes and there is a directed edge from node P1 to mode P2 if P1 is blocked and is waiting for P2 to release some resource. A system is deadlocked if and only if there exists a directed cycle or knot in the WFG. Figure 10.1 shows

a WFG, where process P11 of site 1 has an edge to process P21 of site 1 and an edge to process P32 of site 2. Process P32 of site 2 is waiting for a resource that is currently held by process P33 of site 3. At the same time process P21 at site 1 is waiting on process P24 at site 4 to release a resource, and so on. If P33 starts waiting on process P24, then processes in the WFG are involved in a deadlock depending upon the request model.

4.3 Preliminaries

4.3.1 Deadlock handling strategies

There are three strategies for handling deadlocks, *viz.*, deadlock prevention, deadlock avoidance, and deadlock detection. Handling of deadlocks becomes highly complicated in distributed systems because no site has accurate knowledge of the current state of the system and because every inter-site communication involves a finite and unpredictable delay. Deadlock prevention is commonly achieved either by having a process acquire all the needed



resources simultaneously before it begins executing or by pre-empting a process that holds the needed resource. This approach is highly inefficient and impractical in distributed systems.

In deadlock avoidance approach to distributed systems, a resource is granted to a process if the resulting global system state is safe (note that a global state includes all the processes and resources of the distributed system). Due to several problems, however, deadlock avoidance is impractical in distributed systems.

Deadlock detection requires an examination of the status of process– resource interactions for the presence of cyclic wait. Deadlock detection in distributed systems seems to be the best approach to handle deadlocks in distributed systems. In this chapter, we limit the discussion to deadlock detection techniques in distributed systems.

4.3.2 Issues in deadlock detection

Deadlock handling using the approach of deadlock detection entails addressing two basic issues: first, detection of existing deadlocks and, second, resolution of detected deadlocks.

Detection of deadlocks

Detection of deadlocks involves addressing two issues: maintenance of the WFG and searching of the WFG for the presence of cycles (or knots). Since, in distributed systems, a cycle or knot may involve several sites, the search for cycles greatly depends upon how the WFG of the system is represented across the system. Depending upon the way WFG information is maintained and the search for cycles is carried out, there are centralized, distributed, and hierarchical algorithms for deadlock detection in distributed systems.

Correctness criteria

A deadlock detection algorithm must satisfy the following two conditions:

• **Progress (no undetected deadlocks)** The algorithm must detect all existing deadlocks in a finite time. Once a deadlock has occurred, the deadlock detection activity should continuously progress until the deadlock is detected. In other words, after all wait-for dependencies for a deadlock have formed, the algorithm should not wait for any more events to occur to detect the deadlock.

• **Safety (no false deadlocks)** The algorithm should not report deadlocks that do not exist (called *phantom or false* deadlocks). In distributed systems where there is no global memory and there is no global clock, it is difficult to design a correct deadlock detection algorithm because sites may obtain an out-of-date and inconsistent WFG of the system. As a result, sites may detect a cycle that never existed but whose different segments existed in the system at different times. This is the main reason why many deadlock detection algorithms reported in the literature are incorrect.

Resolution of a detected deadlock

Deadlock resolution involves breaking existing wait-for dependencies between the processes to resolve the deadlock. It involves rolling back one or more deadlocked processes and assigning their resources to blocked processes so that they can resume execution. Note that several deadlock detection algorithms propagate information regarding wait-for dependencies along the edges of the wait-for graph. Therefore, when a wait-for dependency is broken, the corresponding information should be immediately cleaned from the system. If this information is not cleaned in a timely manner, it may result in detection of phantom deadlocks. Untimely and inappropriate cleaning of broken wait-for dependencies is the main reason why many deadlock detection algorithms reported in the literature are incorrect.

4.4 Models of deadlocks

Distributed systems allow many kinds of resource requests. A process might require a single resource or a combination of resources for its execution. This section introduces a hierarchy of request models starting with very restricted forms to the ones with no restrictions

whatsoever. This hierarchy shall be used to classify deadlock detection algorithms based on the complexity of the resource requests they permit.

4.4.1 The single-resource model

The single-resource model is the simplest resource model in a distributed system, where a process can have at most one outstanding request for only one unit of a resource. Since the maximum out-degree of a node in a WFG for the single resource model can be 1, the presence of a cycle in the WFG shall indicate that there is a deadlock. In a later section, an algorithm to detect deadlock in the single-resource model is presented.

4.4.2 The AND model

In the AND model, a process can request more than one resource simultaneously and the request is satisfied only after all the requested resources are granted to the process. The requested resources may exist at different locations. The out degree of a node in the WFG for AND model can be more than 1. The presence of a cycle in the WFG indicates a deadlock in the AND model. Each node of the WFG in such a model is called an AND node. Consider the example WFG described in the Figure 10.1. Process P11 has two outstanding resource requests. In case of the AND model, P11 shall become active from idle state only after both the resources are granted. There is a cycle P11
→P21
→P24
→P54
→P11, which corresponds to a deadlock situation.

In the AND model, if a cycle is detected in the WFG, it implies a deadlock but not vice versa. That is, a process may not be a part of a cycle, it can still be deadlocked. Consider process *P*44 in Figure.1. It is not a part of any cycle but is still deadlocked as it is dependent on *P*24, which is deadlocked. Since in the single-resource model, a process can have at most one outstanding request, the AND model is more general than the single-resource model.

4.4.3 The OR model

In the OR model, a process can make a request for numerous resources simultaneously and the request is satisfied if any one of the requested resources is granted. The requested resources may exist at different locations. If all requests in the WFG are OR requests, then the nodes are called OR nodes. Presence of a cycle in the WFG of an OR model does not imply a deadlock in the OR model. To make it more clear, consider Figure 1. If all nodes are OR nodes, then process P11 is not deadlocked because once process P33 releases its resources, P32 shall become active as one of its requests is satisfied. After P32 finishes execution and releases its resources, process P11 can continue with its processing.

In the OR model, the presence of a knot indicates a deadlock. In a WFG, a vertex v is in a knot if for all u :: u is reachable from v : v is reachable from u. No paths originating from a knot shall have dead ends. A deadlock in the OR model can be intuitively defined as follows:
a process Pi is blocked if it has a pending OR request to be satisfied. With every blocked process, there is an associated set of processes called dependent set. A process shall move from an *idle* to an *active* state on receiving a grant message from any of the processes in its dependent set. A process is permanently blocked if it never receives a grant message from

any of the processes in its dependent set. Intuitively, a set of processes S is deadlocked if all the processes in S are permanently blocked. To formally state that a set of processes is deadlocked, the following conditions hold true:

1. Each of the process is the set S is blocked.
2. The dependent set for each process in S is a subset of S.
3. No grant message is in transit between any two processes in set S.

We now show that a set of processes S shall remain permanently blocked in the OR model if the above conditions are met. A blocked process P is the set S becomes *active* only after receiving a grant message from a process in its dependent set, which is a subset of S. Note that no grant message can be expected from any process in S because they are all blocked. Also, the third condition states that no grant messages in transit between any two processes in set S. So, all the processes in set S are permanently blocked.

Hence, deadlock detection in the OR model is equivalent to finding knots in the graph. Note that, there can be a deadlocked process that is not a part of a knot. Consider Figure 1, where P44 can be deadlocked even though it is not in a knot. So, in an OR model, a blocked process P is deadlocked if it is either in a knot or it can only reach processes on a knot.

4.4.4 The AND-OR model

A generalization of the previous two models (OR model and AND model) is the AND-OR model. In the AND-OR model, a request may specify any combination of *and* and *or* in the resource request. For example, in the ANDOR model, a request for multiple resources can be of the form x *and* (y *or* z). The requested resources may exist at different locations. To detect the presence of deadlocks in such a model, there is no familiar construct of graph theory using WFG. Since a deadlock is a stable property (i.e., once it exists, it does not go away by itself), this property can be exploited and a deadlock in the AND-OR model can be detected by repeated application of the test for OR-model deadlock. However, this is a very inefficient strategy.

4.4.5 The $\binom{p}{q}$ model

Another form of the AND-OR model is the $\binom{p}{q}$ model (called the P-out-of-Q model), which allows a request to obtain any *k* available resources from a pool of n resources. Both the models are the same in expressive power. However, $\binom{p}{q}$ model lends itself to a much more compact formation of a request. Every request in the $\binom{p}{q}$ model can be expressed in the AND-OR model and vice-versa. Note that AND requests for p resources can be stated as $\binom{p}{q}$ and OR requests for p resources can be stated as $\binom{p}{q}$.

4.4.6 Unrestricted model

In the unrestricted model, no assumptions are made regarding the underlying structure of resource requests. In this model, only one assumption that the deadlock is stable is made and hence it is the most general model. This way of looking at the deadlock problem helps in separation of concerns: concerns about properties of the problem (stability and deadlock) are

separated from underlying distributed systems computations (e.g., message passing versus synchronous communication). Hence, these algorithms can be used to detect other stable properties as they deal with this general model. But, these algorithms are of more theoretical value for distributed systems since no further assumptions are made about the underlying distributed systems computations which leads to a great deal of overhead (which can be avoided in simpler models like AND or OR models).

4.5 Knapp's classification of distributed deadlock detection algorithms

Distributed deadlock detection algorithms can be divided into four classes: path-pushing, edge-chasing, diffusion computation, and global state detection.

4.5.1 Path-pushing algorithms

In path-pushing algorithms, distributed deadlocks are detected by maintaining an explicit global WFG. The basic idea is to build a global WFG for each site of the distributed system. In this class of algorithm, whenever deadlock computation is performed, each site sends its local WFG to all the neighboring sites. After the local data structure of each site is updated, this updated WFG is then passed along to other sites, and the procedure is repeated until one site has a sufficiently complete picture of the global state to announce deadlock or to establish that no deadlocks are present. This feature of sending around the paths of the global WFG has led to the term path-pushing algorithms.

4.5.2 Edge-chasing algorithms

In an edge-chasing algorithm, the presence of a cycle in a distributed graph structure is verified by propagating special messages called probes along the edges of the graph. These probe messages are different to the request and reply messages. The formation of a cycle can be detected by a site if it receives the matching probe sent by it previously. Whenever a process that is executing receives a probe message, it simply discards this message and continues. Only blocked processes propagate probe messages along their outgoing edges. An interesting variation of this method can be found in Mitchell, where probes are sent upon request and in the opposite direction of the edges.

The main advantage of edge-chasing algorithms is that probes are fixed size messages that are normally very short.

4.5.3 Diffusing computation-based algorithms

In *diffusion computation*-based distributed deadlock detection algorithms, deadlock detection computation is diffused through the WFG of the system. These algorithms make use of echo algorithms to detect deadlocks. This computation is superimposed on the underlying distributed computation. If this computation terminates, the initiator declares a deadlock. The main feature of the superimposed computation is that the global WFG is implicitly reflected in the structure of the computation. The actual WFG is never built explicitly. To detect a deadlock, a process sends out query messages along all the outgoing edges in the WFG. These queries are successively propagated (i.e., diffused) through the edges of the WFG. Queries are discarded by a running process and are echoed back by blocked processes in the following way: when a blocked process first receives a query message for a particular

deadlock detection initiation, it does not send a reply message until it has received a reply message for every query it sent (to its successors in the WFG). For all subsequent queries for this deadlock detection initiation, it immediately sends back a reply message. The initiator of a deadlock detection detects a deadlock when it has received a reply for every query it has sent out.

4.5.4 Global state detection-based algorithms

Global state detection-based deadlock detection algorithms exploit the following facts: (i) a consistent snapshot of a distributed system can be obtained without freezing the underlying computation, and (ii) a consistent snapshot may not represent the system state at any moment in time, but if a stable property holds in the system before the snapshot collection is initiated, this property will still hold in the snapshot.

Therefore, distributed deadlocks can be detected by taking a snapshot of the system and examining it for the condition of a deadlock.

4.7 Chandy–Misra–Haas algorithm for the AND model

We now discuss Chandy–Misra–Haas's distributed deadlock detection algorithm for the AND model, which is based on edge-chasing. The algorithm uses a special message called *probe*, which is a triplet (i, j, k), denoting that it belongs to a deadlock detection initiated for process P_i and it is being sent by the home site of process P_j to the home site of process P_k. A probe message travels along the edges of the global WFG graph, and a deadlock is detected when a probe message returns to the process that initiated it. A process P_j is said to be *dependent* on another process P_k if there exists a sequence of processes P_j , P_{i1}, P_{i2} , . . . , P_{im}, P_k such that each process except P_k in the sequence is blocked and each process, except the P_j , holds a resource for which the previous process in the sequence is waiting. Process Pj is said to be *locally dependent* upon process P_k if P_j is dependent upon P_k and both the processes are on the same site.

Data structures

Each process Pi maintains a boolean array, dependent i, where dependent (i,j) is true only if P_i knows that P_j is dependent on it. Initially, dependent(i,j) is false for all i and j.

The algorithm

Algorithm is executed to determine if a blocked process is deadlocked. Therefore, a probe message is continuously circulated along the edges of the global WFG graph and a deadlock is detected when a probe message returns to its initiating process.

---

if P_i is locally dependent on itself
then declare a deadlock
else for all P_j and P_k such that
(a) P_i is locally dependent upon P_j, and
(b) P_j is waiting on P_k, and
(c) P_j and P_k are on different sites,
send a probe (i, j, k) to the home site of P_k

On the receipt of a probe (i, j, k), the site takes
the following actions:

if
(d) P_k is blocked, and
(e) $dependent_k(i)$ is false, and
(f) P_k has not replied to all requests P_j,
then
begin
$dependent_k(i) = true$;
if $k = i$
then declare that P_i is deadlocked
else for all P_m and P_n such that
(a′) P_k is locally dependent upon P_m, and
(b′) P_m is waiting on P_n, and
(c′) P_m and P_n are on different sites,
send a probe (i, m, n) to the home site of P_n
end.

Performance analysis

In the algorithm, one probe message (per deadlock detection initiation) is sent on every edge of the WFG which connects processes on two sites. Thus, the algorithm exchanges at most m(n−1)/2 messages to detect a deadlock that involves m processes and spans over n sites. The size of messages is fixed and is very small (only three integer words). The delay in detecting a deadlock is O(n).

4.8 Chandy–Misra–Haas algorithm for the OR model

We now discuss Chandy–Misra–Haas's distributed deadlock detection algorithm for the OR model, which is based on the approach of diffusion computation. A blocked process determines if it is deadlocked by initiating a diffusion computation. Two types of messages are used in a diffusion computation: *query*(i, j, k) and *reply*(i, j, k), denoting that they belong to a diffusion computation initiated by a process Pi and are being sent from process P_j to process P_k.

Basic idea

A blocked process initiates deadlock detection by sending query messages to all processes in its dependent set (i.e., processes from which it is waiting to receive a message). If an active

process receives a *query* or *reply* message, it discards it. When a blocked process Pk receives a *query*(i, j, k) message, it takes the following actions:

1. If this is the first *query* message received by P_k for the deadlock detection initiated by Pi (called the *engaging query*), then it propagates the *query* to all the processes in its dependent set and sets a local variable $num_k(i)$ to the number of *query* messages sent.
2. If this is not the engaging *query*, then P_k returns a *reply* message to it immediately provided P_k has been continuously blocked since it received the corresponding engaging *query*. Otherwise, it discards the *query*.

Process P_k maintains a boolean variable $wait_k(i)$ that denotes the fact that it has been continuously blocked since it received the last engaging *query* from process P_i. When a blocked process P_k receives a *reply*(i, j, k) message, it decrements $num_k(i)$ only if $wait_k(i)$ holds. A process sends a reply message in response to an engaging *query* only after it has received a *reply* to every *query* message it has sent out for this engaging *query*. The initiator process detects a deadlock when it has received *reply* messagesto all the *query* messages it has sent out.

The algorithm

The algorithm works as shown in Algorithm. For ease of presentation, we have assumed that only one diffusion computation is initiated for a process. In practice, several diffusion computations may be initiated for a process (a diffusion computation is initiated every time the process gets blocked), but at any time only one diffusion computation is current for any process. However, messages for outdated diffusion computations may still be in transit. The current diffusion computation can be distinguished from outdated ones by using sequence numbers.

Initiate a diffusion computation for a blocked process P_i:
send *query*(i, i, j) to all processes P_j in the dependent set DS_i of P_i;
$num_i(i) := |DS_i|$; $wait_i(i) := true$;

When a blocked process P_k receives a *query*(i, j, k):
if this is the engaging *query* for process P_i then
send *query*(i, k, m) to all P_m in its dependent set DS_k;
$num_k(i) := |DS_k|$; $wait_k(i) := true$
else if $wait_k(i)$ then send a *reply*(i, k, j) to P_j.

When a process P_k receives a *reply*(i, j, k):
if $wait_k(i)$ then
$num_k(i) := num_k(i) - 1$;
if $num_k(i) = 0$ then
if $i = k$ then **declare a deadlock**
else send *reply*(i, k, m) to the process P_m
which sent the engaging query.

Performance analysis

For every deadlock detection, the algorithm exchanges e *query* messages and e *reply* messages, where e = n(n−1) is the number of edges.

4.9 Global predicate detection

4.9.1 Stable and unstable predicates

Specifying predicates on the system state provides an important handle to specify, observe, and detect the behavior of a system. This is useful in formally reasoning about the system behavior. By being able to detect a specified predicate in the execution, we gain the ability to monitor the execution. Predicate specification and detection has uses in distributed debugging, sensor networks used for sensing in various applications, and industrial process control. As an example in the manufacturing process, a system may be monitoring the pressure of Reagent A and the temperature of Reagent B. Only when 1 = (PressureA > 240 KPa) ∧ (TemperatureB > 300 'C) should the two reagents be mixed. As another example, consider a distributed execution where variables x, y, and z are local to processes Pi, Pj , and Pk, respectively. An application might be interested in detecting the predicate $\varphi 2$=xi+yj+zk <−125. In a nuclear power plant, sensors at various locations would monitor the relevant parameters such as the radioactivity level and temperature at multiple locations within the reactor.

Observe that the "predicate detection" problem is inherently different from the global napshot problem. A global snapshot gives one of the possible states that *could have existed* during the period of the snapshot execution. Thus, a snapshot algorithm can observe only one of the predicate values that *could have existed* during the algorithm execution. Predicates can be either stable or unstable. A *stable* predicate is a predicate that remains true once it becomes true. In traditional systems, a predicate is stable if $\emptyset \Rightarrow \emptyset$, where "$\emptyset$" is the "henceforth" operator from temporal logic. In distributed executions, a more precise definition is needed, due to the absence of global time. Formally, a predicate at a cut C is stable if the following holds:

$$(C \models \phi) \implies (\forall C' \mid C \subseteq C', C' \models \phi).$$

Deadlock in a system is a stable property because the deadlocked processes continue to remain deadlocked (until deadlock resolution is performed). Termination of an execution is another stable property. Specific algorithms to detect termination of the execution, and to detect deadlock were considered. Here, we look at a general technique to detect a stable predicate.

4.9.1 Stable predicates

Deadlock

A deadlock represents a system state where a subset of the processes are blocked on one another, waiting for a reply from the other processes in that subset. The waiting relationship is represented by a wait-for graph (WFG) where an edge from i to j indicates that process i is waiting for a reply from process j. Given a wait-for graph G = (V,E), a *deadlock* is a subgraphG' = (V ',E') such that V' ⊆ V and E' ⊆ E and for each process i in V', theprocess i

remains blocked unless it receives a reply from some process(es) in V'. There are two conditions that characterize the deadlock state of the execution:

• (local condition:) each deadlocked process is locally blocked, and

• (global condition:) the deadlocked process will not receive a reply from some process(es) in V'.

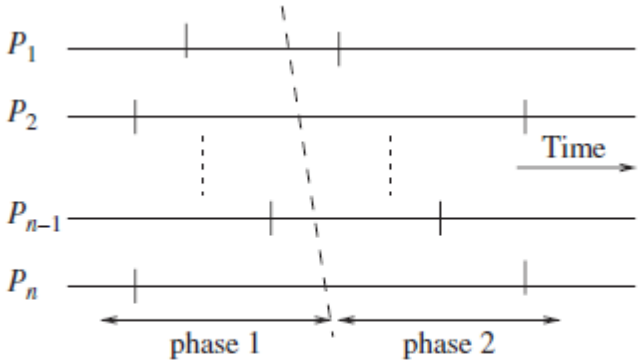
Termination

Termination of an execution is another stable property, and is best understood by viewing a process as alternating between two states: *active state* and *passive state*. An *active* process spontaneously becomes *passive* when it has no further work to do; a *passive process* can become *active* only when it receives a message from some other process. If such a message arrives, then the process becomes *active* by doing CPU processing and maybe sending messages as a result of the processing. An execution is *terminated* if each process is *passive*, and will not become active unless it receives more messages. There are two conditions that characterize the termination state of the execution:

• (local condition:) each process is in passive state; and

• (global condition:) there is no message in transit between any pair of processes.

Generalizing from the above two most frequently encountered stable properties, we assume that each stable property can be characterized by a local process state component, and a channel component or a global component. Recall from our discussion of global snapshots that any channel property can be observed by observing the local states at the two endpoints of the channel, in a consistent manner. Thus, any global condition can be observed by observing the local states of the processes.



Event at which local variables are sampled

We now address the question: "What are the most effective techniques for detecting a stable property?" Clearly, repeatedly or periodically taking a global snapshot will work; if the property is true in some snapshot, then it can be claimed that the property is henceforth true. However, recording a snapshot is expensive; recall that it can require up to $O(n^2)$ control messages without inhibition, or O(n) messages with inhibition. The approach that has been widely adopted is the two-phase approach of observing potentially inconsistent global states. In each state observation, all the local variables necessary for defining the local conditions, as well as the global conditions, are observed.

Two potentially inconsistent global states are recorded consecutively, such that the second recording is initiated after the first recording has completed. This is illustrated in Figure. The stable property can be declared to be true if the following holds:

• The variables on which the local conditions as well as the global conditions are defined have not changed in the two observations, as well as between the two observations.

If none of the variables changes between the two observations, it can be claimed that after the termination of the first observation and before the start of the second observation, there is an instant in physical time when the variables still have the same value. Even though the two observations are each inconsistent, if the global property is true at a common physical time, then the stable property will necessarily be true. The most common ways of taking a pair of consecutive, not necessarily consistent, snapshots using O(n) control messages are as follows:

• Each process randomly records its state variables and sends them to a central process via control messages. When the central process receives this message from each other process, the central process informs each other process to send its (uncoordinated) local state again.

• A token is passed around a ring, and each process appends its local state to the contents of the token. When the token reaches the initiator, it passes the token around for a second time. Each process again appends its local state to the contents of the token.

• On a predefined spanning tree, the root (coordinator) sends a query message in the fan-out sweep of the tree broadcast. In the fan-in sweep of the ensuing tree converge cast, each node collects the local states of the nodes in its subtree rooted at itself and forwards these local states to its parent. When the root gets the local states from all the nodes in its tree, the first phase completes. The second phase, which contains another broadcast followed by a converge cast, is initiated.

4.9.2 Unstable predicates

An *unstable* predicate is a predicate that is not stable and hence may hold only intermittently. The following are some of the several challenges in detecting unstable predicates:

• Due to unpredictable message propagation times and unpredictable scheduling of the various processes on the processors under various load conditions, even for deterministic executions, multiple executions of the same distributed program may pass through different global states. Further, the predicate may be true in some executions and false in others.

• Due to the non-availability of instantaneous time in a distributed system: – even if a monitor finds the predicate to be true in a global state, it may not have actually held in the execution;– even if a predicate is true for a transient period, it may not be detected by intermittent monitoring.

Hence, periodic monitoring of the execution is not adequate. These challenges are faced by snapshot-based algorithms as well as by a central monitor that evaluates data collected from the monitored processes. To address these challenges, we can make two important observations.

• It seems necessary to examine all the states that arise in the execution, so as not to miss the predicate being true. Hence, it seems useful to define predicates, not on individual states, but on the observation of the entire execution.

• For the same distributed program, even given that it is deterministic, multiple observations may pass through different global states. Further, a predicate may be true in some of the

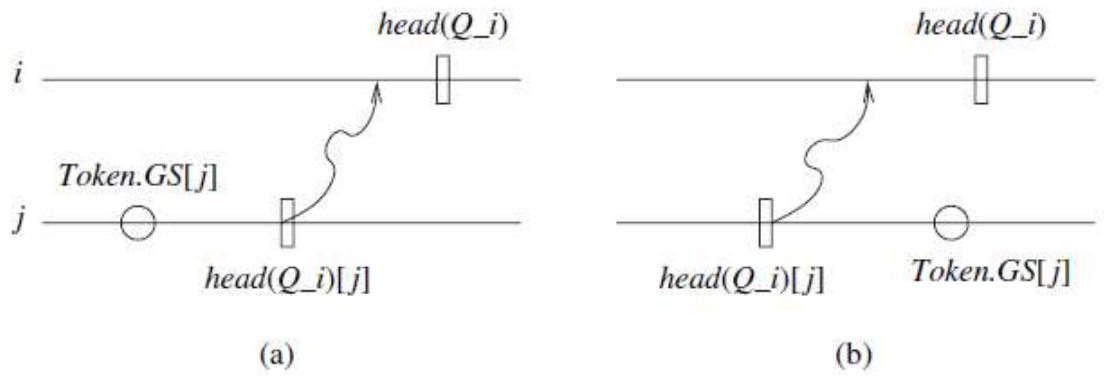
program observations but not in others. Hence it is more useful to define the predicates on all the observations of the distributed program and not just on a single observation of it.

4.10 Distributed algorithms for conjunctive predicates

4.10.1 Distributed state-based token algorithm for Possibly___, where _ is conjunctive

Algorithm is a distributed version of Algorithm. Each queue Qi is maintained locally at Pi. The data structure GS no longer needs to be a n×n array. Instead, a unique token is passed among the processes serially. The token carries a vector GS corresponding to the vector timestamp of the earliest global state under consideration as a candidate solution.

```
struct token {
    integer: GS[1...n];   //Earliest possible global state as a
                          //candidate solution
    boolean: Valid[1...n]; }Token;   //Valid[j] = 0 indicates
                                     //P_j's state GS[j] is invalid
queue of array of integer: Q_i ⟵ ⊥;

Initialization. Token is at a randomly chosen process.
On receiving Token at P_i:
(1)  while (Token.Valid[i] = 0) do   //Token.GS[i] is the latest state of P_i
                                     //known to be inconsistent
(2)      await (Q_i to be nonempty);   //with other candidate local
                                       //state of P_j, for some j
(3)      if ((head(Q_i))[i] > Token.GS[i]) then
(4)          Token.GS[i] ⟵ (head(Q_i))[i];   //earliest possible
                                             //state of P_i that can be part of
                                             //solution is written
(5)          Token.Valid[i] ⟵ 1;   //to Token and its validity is set.
(6)      else dequeue head(Q_i);
(7)  for j = 1 to n (j ≠ i) do   //for each other process P_j: based on P_i's
                                 //local state, determine whether
(8)      if j ≠ i and (head(Q_i))[j] ≥ Token.GS[j] then   //P_j's
                                                          //candidate local state (in Token)
                                                          //is consistent. If not, P_j needs to
                                                          //consider a later candidate
(9)          Token.GS[j] ⟵ (head(Q_i))[j];   // state with a
                                             //timestamp > (head(Q_i))[j]
(10)         Token.Valid[j] ⟵ 0;
(11) dequeue (headQ_i);
(12) if for some k, Token.Valid[k] = 0 then
(13)     send Token to P_k;
(14) else return(1).
```


head(Q_i)
i
Token.GS[j]
j
head(Q_i)[j]
(a)
head(Q_i)
head(Q_i)[j]
Token.GS[j]
(b)

Karpagam Academy of Higher Education
Department of Computer Applications
Subject Code/Name : Elective:17CAP505N/Distributed Computing

Objective Type Questions

	UNIT 4							
S.No	**Questions**	**OPTION 1**	**OPTION 2**	**OPTION 3**	**OPTION 4**	**OPTION 5**	**OPTION 6**	**Answer Key**
1	_____are a fundamental problem in DS.	Deadlocks	processors	task	procedures			**Deadlocks**
2	Deadlock is commo_____nly achieved by either having a process acquire all the needed resources simultaneously before it begins execution or by pre empting a process than holds the needed resources.	acceptance	prevention	detection	looping			**prevention**
3	In the _____ to DS a resource is granted to a process if the resulting global system is safe.	deadlock condition	Detection	deadlock avoidance approach	acceptance			**deadlock avoidance approach**
4	Deadlock detection requires an examination of the status of the process resources interaction for the presence of a _____.	deadlock detection	deadlock acceptance	deadlock resources	deadlock condition			**deadlock condition**
5	A DS consists of a set of processors that are connected by _____.	protocols	resources	communication networks	layers			**communication networks**
6	The communication delays is finite but _____.	predictable	unpredictable	limited	unlimited			**unpredictable**
7	A _____ can be running state if a process has all the needed resources and is either executing or is ready for execution.	Process	tasks	procedures	functions			**Process**
8	In the blocked state a process is waiting to acquire some_____	Process	Methods	resources.	Procedures			**resources.**
9	In DS the state of the system can be modelled by directed graph called a _____	DFG	Graph	Wait and accept	wait for graph.			**wait for graph.**
10	A system is _____ if and only if there exists a directed cycle or knot in the WFG.	terminates	accepted	deadlocked	leaked			**deadlocked**
11	Handling of ____ becomes highly complicated in DS because no site has accurate knowledge of the current state of the system.	tasks	deadlocks	process	resources			**deadlocks**
12	In _____ to DS a resource is granted to a process if the resulting global system state is safe.	deadlock avoidance approach	deadlock condition	deadlock acceptance	communication networks			**deadlock avoidance approach**
13	_____ in DS seems to be the best approach to handle deadlocks in DS.	deadlock avoidance approach	Deadlock detection	deadlock acceptance	communication networks			**Deadlock detection**
14	_____ requires an examination of the status of the process resources interaction for the presence of a cycle wait.	deadlock avoidance approach	deadlock acceptance	Deadlock detection	communication networks			**Deadlock detection**

15	Deadlock handling using the approach of deadlock detection entails _____ two basic issues.	packeting	terminating	detecting	addressing			**addressing**
16	_____ must detect all existing deadlocks in a finite time.	Primary algorithm	safety algorithm	Progress algorithm	Process algorithm			**Progress algorithm**
17	_____ should not report deadlocks that do not exists called phantom or false deadlocks.	Primary algorithm	Safety algorithm	Progress algorithm	Process algorithm			**Safety algorithm**
18	_____ involves breaking existing wait for dependencies between the processes to resolve the deadlock.	Deadlock resolution	Safety algorithm	Progress algorithm	Process algorithm			**Deadlock resolution**
19	_____detection algorithms propagate information regarding wait for graph.	Deadlock resolution	Several deadlocks	detecting	addressing			**Several deadlocks**
20	The_____ is not cleaned in a timely manner, it am result in detection of phantom deadlocks.	data	tasks	information	process			**information**
21	DS allow many kinds of_____.	resouce detection	resource utilization	resource acceptance	resource requests			**resource requests**
22	A process might require a _____ or a combination of resources for its execution.	multi resources	waited resources	single resource	unwaited resources			**single resource**
23	The _____ is the simplest resource model in a DS where process can have almost one outstanding request for only one unit of a resource.	multi resource model	single resource model	waited resources	unwaited resources			**single resource model**
24	In the _____ a process can request more than one resource simultaneously and the request is satisfied only after all the requested resources are granted to the process.	AND model	OR model	AND OR model	WFG			**AND model**
25	In the AND model if a cycle is detected in the _____ it implies a deadlock but not vice versa.	AND Model	WFG	OR model	AND OR model			**WFG**
26	In the _____ a process can make a request for numerous resources simultaneously and the request is satisfied if any one of the requested resources is granted.	AND Model	WFG	OR model,	AND OR model			**OR model,**
27	The _____may exist at different locations.	resouce detection	resource utilization	resource acceptance	requested resources			**requested resources**
28	Deadlock detection in the OR model is equivalent to find _____ in the graph.	data	tasks	knots	process			**knots**
29	A generalization of the previous models is the_____.	AND_OR MODEL	AND model	OR model	WFG			**AND_OR MODEL**
30	Another form of the AND_OR model is the model also called as _____- model.	P	P out of Q	Q	P of q			**P out of Q**
31	In the _____, no assumptions are made regarding the underlying structure of resource requests.	restricted model	complex model	unrestricted model	static model			**unrestricted model**
32	In _____ algorithms distributed deadlocks are detected by maintaining are explicit global WFG.	path identification	path determination	path detection	path pushing			**path pushing**
33	In an edge chasing algorithm the presence of a cycle in a _____-structure is verified by propagating special messages called probes along the edges of the graph.	distributed task	distributed process	distributed graph	distributed networks			**distributed graph**

34	The main advantage of edge chasing algorithms is that probes are _____ size messages that are normally very short.	variables	fixed	same	different			**fixed**
35	In diffusion computation based distributed deadlock detection algorithms deadlock detection computation is diffused through the _____ of the system.	WFG	AND model	OR model	AND OR model			**WFG**
36	To detect the deadlock, a process sends out query messages along all the outgoing edges into the _____-.	AND model	WFG	OR model	AND OR model			**WFG**
37	A consistent snapshot of a DS can be obtained without _____ the underlying computation.	terminating	accepting	freezing	processing			**freezing**
38	Mitchell and Merritt's algorithm belongs to the class of _____ algorithms where probes are sent in the opposite direction to the edges of the WFG.	local processing	global processing	edge clustering	edge chasing			**edge chasing**
39	A private label which is unique to the node at all times though it is not _____.	constant	local initialization	global initialization	variable			**constant**
40	A _____ which can be read by the other processes and which may not be unique.	private label	public label	protected label	processed label			**public label**
41	A global WFG is maintained and it defines the _____of the system.	localstate	global state	entire state	single state			**entire state**
42	Chandy Misra Haas distributed deadlock detection algorithm for the _____ which is based on edge chasing.	OR model	AND OR model	WFG	AND model			**AND model**
43	Chandy Misra Haas distributed deadlock detection algorithm for the _____which is based on approach of diffusion computation.	AND OR model	WFG	OR model	AND model			**OR model**
44	A blocked process initiates _____ by sending query messages to all processes in its dependent set.	deadlock avoidance	deadlock processing	deadlock accepting	deadlock detection			**deadlock detection**
45	The current _____ can be distinguished from outdated ones by using sequence numbers.	diffusion termination	diffusion detection	diffusion computation	all diffusion			**diffusion computation**
46	The Kshemkalyani Singhal algorithm to _____ deadlocks in the P out of the Q model is based on the global state detection approach.	computer	detect	termiante	stacks			**detect**
47	A _____- snapshot gives one of the possible states that could have existed during the period of the snapshot execution.	global	local	subset	set			**global**
48	A stable predicte is a predicate that remains true once it becomes _____.	FALSE	TRUE	may be	legal			**TRUE**
49	In distributed executions a more precise definition is needed due to the absence of _____.	local time	accessing time	global time	terminating time			**global time**
50	A deadlock represents a system state where a subset of the processes are blocked on one another waiting fro a reply from the other processes in that _____.	set	stack	unstack	subset			**subset**

51	An_____ spontaneously becomes active only when it receives a message from some other processes.	passive process	local access	global access	active process			**active process**
52	A _____ can become active only when it receives a message from some other process.	local access	global access	passive process	active process			**passive process**
53	An _____ is terminated if each process is passive and will not become active unless it receives more messages.	termination	execution	processing	stoping			**execution**
54	An _____-is a predicate that is not stable and hence may hold only intermittenly.	unstable predicate	stable predicate	locally started	globally started			**unstable predicate**
55	A unique token is passed among the processes serially in the _____.	local algorithms	global algorithms	processing algorithms	distributed algorithms			**distributed algorithms**
56	A total of m states at a process the time overhead at a process is_____.	O(m)	O(n)	O(mn)	O(2mn)			**O(mn)**
57	The _____across processes is cumulative as the token travels serially.	time lapsed	time overhead	time delay	time batching			**time overhead**
58	The total time complexity is _____	$O(mn)^2$	O(n)	O(mn)	O(2mn)			**$O(mn)^2$**
59	Across all the processes the space requirement becomes _____	O(n)	$O(mn)^2$	O(mn)	O(2mn)			**$O(mn)^2$**
60	The token makes O(mn) hops and the size of the token is _____.	2n integers	n integers	mn integers	2m integers			**2n integers**

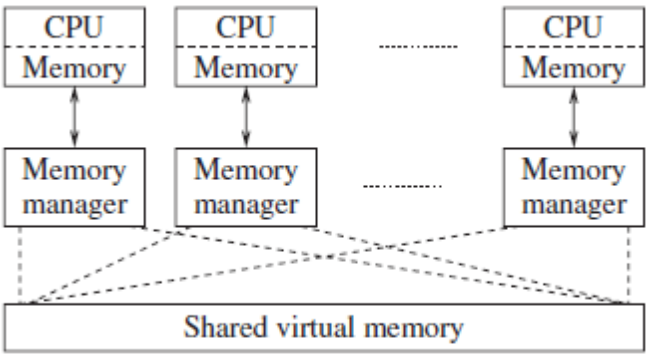
5.1 Distributed shared memory

5.1.1 Abstraction and advantages

Distributed shared memory (DSM) is an abstraction provided to the programmer of a distributed system. It gives the impression of a single monolithic memory, as in traditional von Neumann architecture. Programmers access the data across the network using only *read* and *write* primitives,as they would in a uniprocessor system. Programmers do not have to deal with *send* and *receive* communication primitives and the ensuing complexity of dealing explicitly with synchronization and consistency in the message passing model. The DSM abstraction is illustrated in Figure. A part of each computer's memory is earmarked for shared space, and the remainder is private memory. To provide programmers with the illusion of a single shared address space, a memory mapping management layer is required to manage the *shared virtual memory* space.

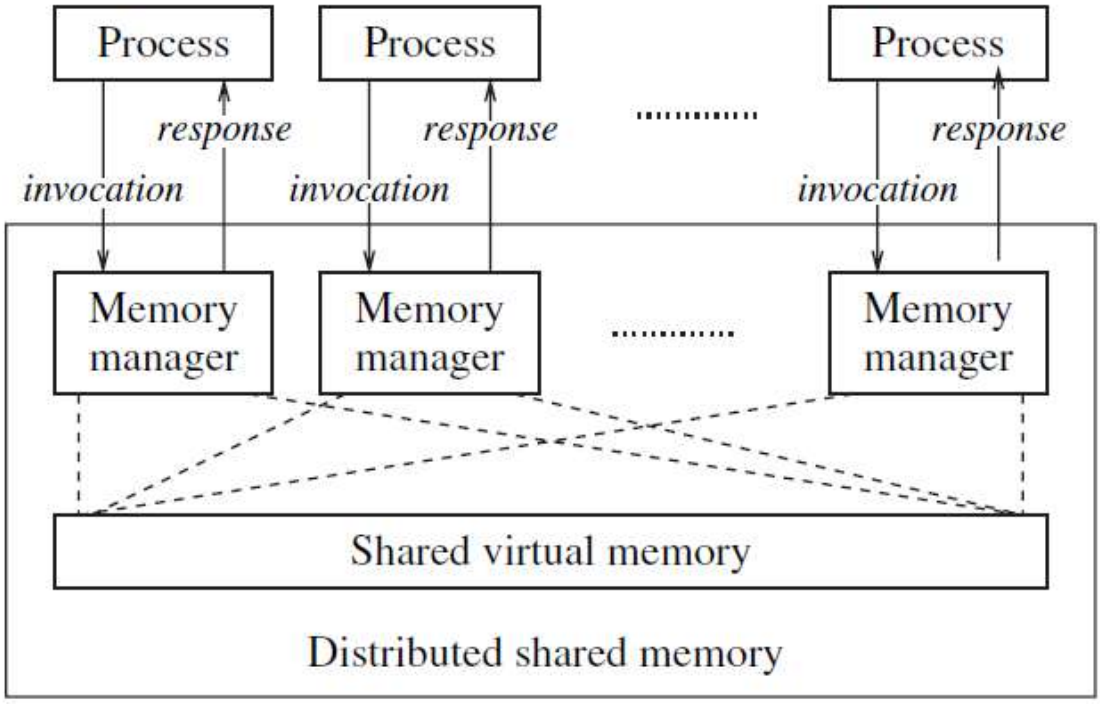
DSM has the following advantages:

1. Communication across the network is achieved by the read/write abstraction that simplifies the task of programmers.
2. A single address space is provided, thereby providing the possibility of avoiding data movement across multiple address spaces, and simplifying *passing-by-reference* and passing complex data structures containing pointers.
3. If a block of data needs to be moved, the system can exploit locality of reference to reduce the communication overhead.
4. DSM is often cheaper than using dedicated multiprocessor systems, because it uses simpler software interfaces and off-the-shelf hardware.
5. There is no bottleneck presented by a single memory access bus.
6. DSM effectively provides a large (virtual) main memory.
7. DSM provides portability of programs written using DSM. This portability arises due to a common DSM programming interface, which is independent of the operating system and other low-level system characteristics.



Although a familiar (i.e., read/write) interface is provided to the programmer there is a catch to it. Under the covers, there is inherently a distributed system and a network, and the data needs to be shared in some fashion. There is no silver bullet. Moreover, with the possibility of data replication and/or the concurrent access to data, concurrency control needs to be

enforced. Specifically, when multiple processors wish to access the same data object, a decision about how to handle concurrent accesses needs to be made. As in traditional databases, if a locking mechanism based on read and write locks for objects is used, concurrency is severely restrained, defeating one of the purposes of having the distributed system. On the other hand, if concurrent access is permitted by different processors to different replicas, the problem of replica consistency (which is a generalization of the problem of cache consistency in computer architecture studies) needs to be addressed. The main point of allowing concurrent access (by different processors) to the same data object is to increase throughput. But in the face of concurrent access, the semantics of what value a read operation returns to the program needs to be specified. Programmers ultimately need to understand this semantics, which may differ from the Von Neumann semantics, because the program logic depends greatly on this semantics. This compromises the assumption that the DSM is transparent to the programmer.



Before examining the challenges in implementing replica coherency in DSM systems, we look at its disadvantages:

1. Programmers are not shielded from having to know about various replica consistency models and from coding their distributed applications according to the semantics of these models.
2. As DSM is implemented under the covers using asynchronous message passing, the overheads incurred are at least as high as those of a message passing implementation. As such, DSM implementations cannot be more efficient than asynchronous message-passing implementations. The generality of the DSM software may make it less efficient.
3. By yielding control to the DSM memory management layer, programmers lose the ability to use their own message-passing solutions for accessing shared objects. It is likely that the

standard vanilla implementations of DSM have a higher overhead than a programmer-written implementation tailored for a specific application and system.

The main issues in designing a DSM system are the following:

• Determining what semantics to allow for concurrent access to shared objects. The semantics needs to be clearly specified so that the programmer can code his program using an appropriate logic.

• Determining the best way to implement the semantics of concurrent access to shared data. One possibility is to use replication. One decision to be made is the degree of replication – partial replication at some sites, or full replication at all the sites. A further decision then is to decide on whether to use read-replication (replication for the read operations) or write-replication (replication for the write operations) or both.

• Selecting the locations for replication (if full replication is not used), to optimize efficiency from the system's viewpoint.

• Determining the location of remote data that the application needs to access, if full replication is not used.

• Reducing communication delays and the number of messages that are involved under the covers while implementing the semantics of concurrent
access to shared data.

There is a wide range of choices on how these issues can be addressed. In part, the solution depends on the system architecture.

There are four broad dimensions along which DSM systems can be classified and implemented:

• Whether data is replicated or cached.

• Whether remote access is by hardware or by software.

• Whether the caching/replication is controlled by hardware or software.

• Whether the DSM is controlled by the distributed memory managers, by the operating system, or by the language runtime system.

Table 12.1 Comparison of DSM systems (adapted from [29]).

Type of DSM	Examples	Management	Caching	Remote access
Single-bus multiprocessor	Firefly, Sequent	by MMU	hardware control	by hardware
Switched multiprocessor	Alewife, Dash	by MMU	hardware control	by hardware
NUMA system	Butterfly, CM*	by OS	software control	by hardware
Page-based DSM	Ivy, Mirage	by OS	software control	by software
Shared variable DSM	Midway, Munin	by language runtime system	software control	by software
Shared object DSM	Linda, Orca	by language runtime system	software control	by software

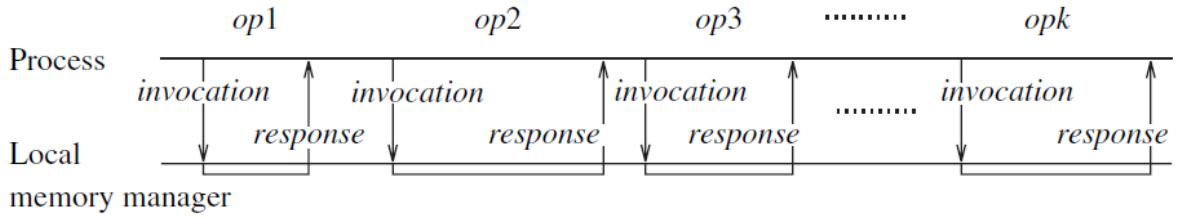
5.1.2 Memory consistency models

Memory coherence is the ability of the system to execute memory operations correctly. Assume n processes and si memory operations per process Pi. Also assume that all the

operations issued by a process are executed sequentially (that is, pipelining is disallowed), as shown in Figure. Observe that there are a total of

$$(s_1 + s_2 + \ldots + s_n)!/(s_1!s_2!\ldots s_n!)$$

possible permutations or interleavings of the operations issued by the processes. The problem of ensuring memory coherence then becomes the problem of identifying which of these interleavings are "correct," which of course requires a clear definition of "correctness." The *memory consistency model* defines the set of allowable memory access orderings. While a traditional definition of correctness says that a correct memory execution is one that returns to each *Read* operation, the value stored by the most recent *Write* operation, the very definition of "most recent" becomes ambigious in the presence of concurrent access and multiple replicas of the data item. Thus, a clear definition of correctness is required in such a system; the objective is to disallow the interleavings that make no semantic sense, while not being overly restrictive so as to permit a high degree of concurrency.



The DSM system enforces a particular memory consistency model; programmers write their programs keeping in mind the allowable interleavings permitted by that specific memory consistency model. A program written for one model may not work correctly on a DSM system that enforces a different model. The model can thus be viewed as a *contract* between the DSM system and the programmer using that system. We now consider six consistency models, which are related as shown in Figure.

Notation A write of value a to variable x is denoted as *Write*(x,a). A read of variable x that returns value a is denoted as *Read*(x,a). A subscript on these operations is sometimes used to denote the processor that issues these operations.

5.1.3 Strict consistency/atomic consistency/linearizability

The strictest model, corresponding to the notion of correctness on the traditional Von Neumann architecture or the uniprocessor machine, requires that any *Read* to a location (variable) should return the value written by the most recent *Write* to that location (variable). Two salient features of such a system are the following: (i) a common global time axis is implicitly available in a uniprocessor system; (ii) each write is immediately visible to all processes.

Adapting this correctness model to a DSM system with operations that can be concurrently issued by the various processes gives the *strict consistency model*, also known as the *atomic consistency model*. The model is more formally specified as follows:

1. Any *Read* to a location (variable) is required to return the value written by the most recent *Write* to that location (variable) as per a global time reference.

For operations that do not overlap as per the global time reference, the specification is clear. For operations that overlap as per the global time reference, the following further specifications are necessary.

2. All operations appear to be executed atomically and sequentially.

3. All processors see the same ordering of events, which is equivalent to the global-time occurrence of non-overlapping events.

An alternate way of specifying this consistency model is in terms of the "invocation" and "response" to each *Read* and *Write* operation, as shown in Figure. Recall that each operation takes a finite time interval and hence different operations by different processors can overlap in time. However, the invocation and the response to each invocation can both be separately viewed as being atomic events. An execution sequence in global time is viewed as a sequence Seq of such invocations and responses. Clearly, Seq must satisfy the following conditions:

• (Liveness:) Each invocation must have a corresponding response.

• (Correctness:) The projection of Seq on any processor i, denoted Seq_i, must be a sequence of alternating invocations and responses if pipelining is disallowed.

Despite the concurrent operations, a linearizable execution needs to generate an equivalent global order on the events that is a permutation of Seq, satisfying the semantics of *linearizability*. More formally, a sequence Seq of invocations and responses is *linearizable* (LIN) if there is a permutation Seq' of adjacent pairs of corresponding (invoc, resp) events satisfying:

1. For every variable v, the projection of Seq' on v, denoted Seq'_v, is such that every *Read* (adjacent (invoc, resp) event pair) returns the most recent *Write* (adjacent (invoc, resp) event pair) that immediately preceded it.

2. If the response op1_resp_ of operation op1 occurred before the invocation op2_invoc_ of operation op2 in Seq, then op1 (adjacent (invoc, resp), event pair) occurs before op2 (adjacent (invoc, resp) event pair) in Seq'.

Condition 1 specifies that every processor sees a common order Seq' of events, and that in this order, the semantics is that each *Read* returns the most recent completed *Write* value. Condition 2 specifies that the common order Seq' must satisfy the global time order of events, viz., the order of non-overlapping operations in Seq must be preserved in Seq'.

Examples Figure shows three executions:

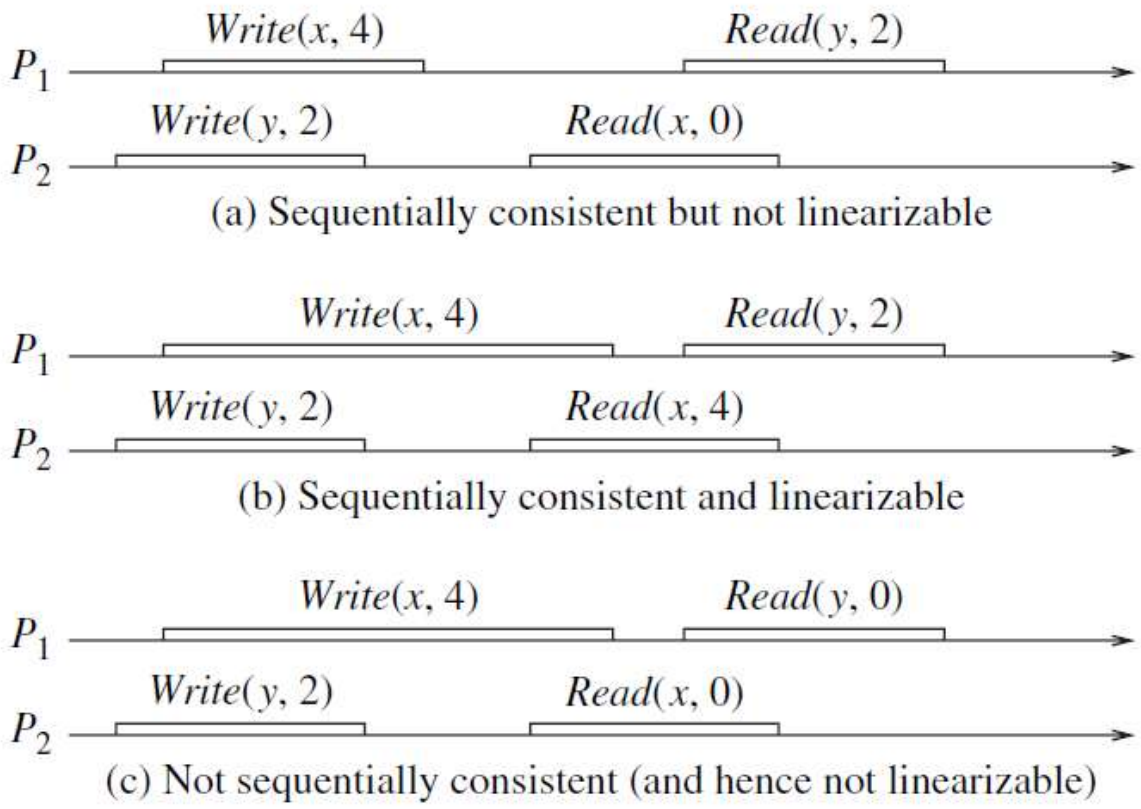
• **Figure (a)** The execution is not linearizable because although the *Read* by P2 begins after *Write*(x,4), the *Read* returns the value that existed before the *Write*. Hence, a permutation Seq' satisfying the condition 2 above on global time order does not exist.

• **Figure (b)** The execution is linearizable. The global order of operations (corresponding to (invocation, response) pairs in Seq'), consistent with the real-time occurrence, is: *Write(y,2)*, *Write*(x,4), *Read*(x,4), *Read*(y,2). This permutation Seq' satisfies conditions 1 and 2.

• **Figure 12.4(c)** The execution is not linearizable. The two dependencies: *Read*(x,0) before *Write*(x,4), and *Read*(y,0) before *Write*(x,2) cannot both be satisfied in a global order while satisfying the local order of operations at each processor. Hence, there does not exist any permutation Seq_ satisfying conditions 1 and 2.

Implementations

Implementing linearizability is expensive because a global time scale needs to be simulated. As all processors need to agree on a common order, the implementation needs to use total order. For simplicity, we assume full replication of each data item at all the processors. Hence, total ordering needs to be combined with a broadcast. Algorithm gives the implementation



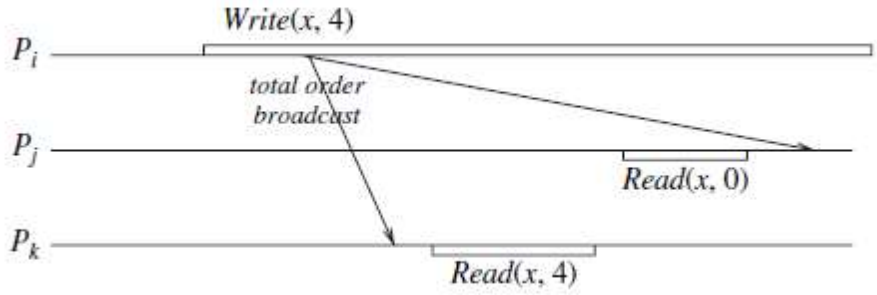
assuming the existence of a *total order broadcast* primitive that broadcasts to all processors including the sender. Hence, the memory manager software has to be placed between the application above it and the total order broadcast layer below it.

```
(shared var)
int: x;
(1)   When the memory manager receives a Read or Write from application:
(1a)  total_order_broadcast the Read or Write request to all processors;
(1b)  await own request that was broadcast;
(1c)  perform pending response to the application as follows
(1d)        case Read: return value from local replica;
(1e)        case Write: write to local replica and return ack to application.

(2)   When the memory manager receives a total_order_broadcast(Write,
      x, val) from network:
(2a)  write val to local replica of x.

(3)   When the memory manager receives a total_order_broadcast(Read,
      x) from network:
(3a)  no operation.
```

Although Algorithm appears simple, it is also subtle. The total order broadcast ensures that all processors see the same order:

• For two non-overlapping operations at different processors, by the very definition of non-overlapping, the response to the former operation precedes the invocation of the latter in global time.

• For two overlapping operations, the total order ensures a common view by all processors.



For a *Read* operation, when the memory managers system wide receive the total order broadcast, they do not perform any action. Why is the broadcast then necessary? The reason is this. If *Read* operations do not participate in the total order broadcasts, they do not get totally ordered with respect to the *Write* operations as well as with respect to the other *Read* operations. This can lead to a violation of linearizability, as shown in Figure. The *Read* by Pk returns the value written by Pi. The later *Read* by Pj returns the initial value of 0. As per the global time ordering requirement of linearizability, the *Read* by Pj that occurs after the *Read* by Pk must also return the value 4. However, that is not the case in this example, wherein the *Read* operations do not participate in the total order broadcast.

5.2 Shared memory mutual exclusion

Operating systems have traditionally dealt with multi-process synchronization using algorithms based on first principles (e.g., the well-known bakery algorithm), high-level constructs such as *semaphores* and *monitors*, and special "atomically executed" instructions supported by special-purpose hardware (e.g., *Test&Set*, *Swap*, and *Compare&Swap*). These algorithms are applicable to all shared memory systems. In this section, we will review the bakery algorithm, which requires O(n) accesses in the entry section, irrespective of the level of contention. We will then study *fast mutual exclusion*, which requires O(1) accesses in the entry section in the absence of contention. This algorithm also illustrates an interesting technique in resolving concurrency. As hardware primitives have the in-built atomicity that helps to easily solve the mutual exclusion problem, we will then examine mutual exclusion based on these primitives.

5.3. Lamport's bakery algorithm

Lamport proposed the classical *bakery algorithm* for n-process mutual exclusion in shared memory systems. The algorithm is so called because it mimics the actions that customers follow in a bakery store. A process wanting to enter the critical section picks a token number that is one greater than the elements in the array choosing[1.. n]. Processes enter the critical section in the increasing order of the token numbers. In case of concurrent accesses to choosing by multiple processes, the processes may have the same token number. In this case, a unique *lexicographic order* is defined on the tuple (token, pid), and this dictates the order in which processes enter the critical section. The algorithm for process i is given in Algorithm. The algorithm can be shown to satisfy the three requirements of the critical section problem: (i) mutual exclusion, (ii) bounded waiting, and (iii) progress. In the entry section, a process chooses a timestamp for itself, and resets it to 0 in the exit section. In lines 1a–1c, each process chooses a timestamp for itself, as the max of the latest timestamps of all processes, plus one. These steps are non-atomic; thus multiple processes could be choosing timestamps in overlapping durations. When process i reaches line 1d, it has to check the status of each other process j, to deal with the effects of any race conditions in selecting timestamps. In lines 1d–1f, process i serially checks the status of each other process j. If j is selecting a timestamp for itself, j's selection interval may have overlapped with that of i, leading to an unknown order of timestamp values. Process i needs to make sure that any other process j (j < i) that had begun to execute line 1b concurrently with itself and may still be executing line 1b does not assign itself the same timestamp. Otherwise mutual exclusion could be violated as i would enter the CS, and subsequently, j, having a lower process identifier and hence a lexicographically lower timestamp, would also enter the CS. Hence, i waits for j's timestamp to stabilize, i.e., choosing(j) to be set to *false*. Once j's timestamp is stabilized, i moves from line 1e to line 1f. Either j is not requesting (in which case j's timestamp is 0) or j is requesting. Line 1f determines the relative priority between i and j. The process with a *lexicographically* lower timestamp has higher priority and enters the CS; the other process has to wait (line 1g). Hence, *mutual exclusion* is satisfied. *Bounded waiting* is satisfied because each other process j can "overtake" process i at most once after i has completed choosing its timestamp. The second time j chooses a timestamp, the value will necessarily be larger than i's timestamp if i has not yet entered its CS. *Progress* is guaranteed because the

lexicographic order is a total order and the process with the lowest timestamp at any time in the loop (lines 1d–1g) is guaranteed to enter the CS.

```
(shared vars)
boolean: choosing[1...n];
integer: timestamp[1...n];
repeat
(1)   P_i executes the following for the entry section:
(1a)  choosing[i] ⟵ 1;
(1b)  timestamp[i] ⟵ max_{k∈[1...n]}(timestamp[k]) + 1;
(1c)  choosing[i] ⟵ 0;
(1d)  for count = 1 to n do
(1e)        while choosing[count] do no-op;
(1f)        while timestamp[count] ≠ 0 and (timestamp[count], count)
              < (timestamp[i], i) do
(1g)              no-op.
(2)   P_i executes the critical section (CS) after the entry section
(3)   P_i executes the following exit section after the CS:
(3a)  timestamp[i] ⟵ 0.
(4)   P_i executes the remainder section after the exit section
until false;
```

• *Space complexity*: A lower bound of n registers, specifically, the timestamp array, has been shown for the shared memory critical section problem. Thus, one cannot hope to have a more space-efficient algorithm for distributed shared memory mutual exclusion.

• *Time complexity*: In many environments, the level of contention may be low. The O(n) overhead of the entry section does not scale well for such environments. This concern is addressed by the field of *fast mutual exclusion* that aims to have O(1) time overhead for the entry and exit sections of the algorithm, in the absence of contention. Although this algorithm guarantees mutual exclusion and progress, unfortunately, this fast algorithm has a price – in the worst case, it does not guarantee bounded delay. Next, we will study Lamport's algorithm for fast mutual exclusion in asynchronous shared memory systems. This algorithm is notable in that it is the first algorithm for fast mutual exclusion, and uses the asynchronous shared memory model. Further, it illustrates an important technique for resolving contention. The worst-case unbounded delay in the presence of persisting contention has been addressed subsequently, by using a timed model of execution, wherein there is an upper bound on the time it takes to execute any step.

5.3.2 Lamport's WRWR mechanism and fast mutual exclusion

Lamport's *fast mutual exclusion* algorithm is given in Algorithm. The algorithm illustrates an important technique – the (W −R−W −R) sequence that is a necessary and sufficient sequence of operations to check for contention and to ensure safety in the entry section, using only two registers.

Lines 1b, 1c, 1g , and 1h represent a basic (W(x)-R(y)-W(y)-R(x))vsequence whose necessity in identifying a minimal sequence of operations for fast mutual exclusion is justified as follows:

1. The first operation needs to be a *Write*, say to variable x. If it were a *Read*, then all contending processes could find the value of the variable even outside the entry section.
2. The second operation cannot be a *Write* to another variable, for that could equally be combined with the first *Write* to a larger variable. The second operation should *not* be a *Read* of x because it follows *Write* of x and if there is no interleaved operation from another process, the *Read* does not provide any new information. So the second operation must be a *Read* of another variable, say y.
3. The sequence must also contain *Read*(x) and *Write*(y) because there is no point in reading a variable that is not written to, or writing a variable that is never read.

```
(shared variables among the processes)
integer: x, y;                                   // shared register initialized
boolean b[1. . . n];    // flags to indicate interest in critical section
repeat
(1)    P_i (1 ≤ i ≤ n) executes entry section:
(1a)       b[i] ⟵ true;
(1b)       x ⟵ i;
(1c)       if y ≠ 0 then
(1d)                b[i] ⟵ false;
(1e)                await y = 0;
(1f)                goto (1a);
(1g)       y ⟵ i;
(1h)       if x ≠ i then
(1i)                b[i] ⟵ false;
(1j)                for j = 1 to n do
(1k)                        await ¬b[j];
(1l)                if y ≠ i then
(1m)                        await y = 0;
(1n)                        goto (1a);
(2)    P_i (1 ≤ i ≤ n) executes critical section:
(3)    P_i (1 ≤ i ≤ n) executes exit section:
(3a)       y ⟵ 0;
(3b)       b[i] ⟵ false;
forever.
```

4. The last operation in the minimal sequence of the entry section must be a *Read*, as it will help determine whether the process can enter CS. So the last operation should be *Read*(x), and the second-last operation should be the *Write*(y). In the absence of contention, each process writes its own i.d. to x and then reads y. Then finding that y has its initial value, the

process writes its own i.d. to y and then reads x. Finding x to still be its own i.d., it enters CS. Correctness needs to be shown in the presence of contention – let us discuss this after considering the structure of the remaining entry and exit section code.

In the exit section, the process must do a *Write* to indicate its completion of the CS. The *Write* cannot be to x, which is also the first variable written in the entry section. So the operation must be *Write*(y).

Now consider the sequence of interleaved operations by processes i, j, and k in the entry section, as shown in Figure. Process i enters its critical

Process P_i	Process P_j	Process P_k	Variables
	$W_j(x)$		$\langle x = j, y = 0 \rangle$
$W_i(x)$			$\langle \mathbf{x = i}, y = 0 \rangle$
$R_i(y)$			$\langle x = i, y = 0 \rangle$
	$R_j(y)$		$\langle x = i, y = 0 \rangle$
$W_i(y)$			$\langle x = i, \mathbf{y = i} \rangle$
	$W_j(y)$		$\langle x = i, y = j \rangle$
$R_i(x)$			$\langle x = i, y = j \rangle$
		$W_k(x)$	$\langle x = k, y = j \rangle$
	$R_j(x)$		$\langle x = k, y = j \rangle$

section, but there is no record of its identity or that it had written any variables at all, because the variables it wrote (shown boldfaced above) have been overwritten. In order that other processes can discover when (and who) leaves the CS, there needs to be another variable that is set before the CS and reset after the CS. This is the boolean, b(i). Additionally, y needs to be reset on exiting the CS.

The code in lines 1c–1f has the following use. If a process p finds y '= 0, then another process has executed at least line 1g and not yet executed line 3a. So process p resets its own flag, and before retrying again, it awaits for y = 0. If process p finds y = 0 in line 1c, it sets y = p in line 1g and checks if x = p.

• If x = p, then no other process has executed line 1b, and any later process would be blocked in the loop in lines 1c–1f now because y = p. Thus, if x = p, process p can safely enter the CS.

• If x '= p, then another process, say q, has overwritten x in line 1b and there is a potential race. Two broad cases are possible:

– Process q finds y '= 0 in line 1c. It resets its flag, and stays in the 1d–1f section at least until p has exited the CS. Process p on the other hand resets its own flag (line 1i) and waits for all other processes such as q to reset their own flags. As process q is trapped in lines 1d–1f, process p will find y = p in line 1l and enter the CS.

– Process q finds y = 0 in line 1c. It sets y to q in line 1g, and enters the race, even closer to process p, which is at line 1h. Of the processes such as p and q that contend at line 1h, there will be a unique winner:

* If no other process r has since written to x in line 1b, the winner is the process among p and q that executed line 1b last, i.e., wrote its own i.d. to x. That winner will enter the CS directly

from line 1h, whereas the losers will reset their own flags, await the winner to exit and reset its flag, and also await other contenders at line 1h and newer contenders to reset their own flags. The losers will compete again from line 1a after the winner has reset y.

* If some other process r has since written its i.d. to x in line 1b, both p and q will enter code in lines 1i–1n. Both p and q reset their flags, await for r, which will be trapped in lines 1d–1f to reset its flag, and then both p and q check the value of y. Between p and q, the process that last wrote to y in line 1g will become the unique winner and enter the CS directly. The loser will then await for the winner to reset y, and then compete again from line 1a.

Thus, mutual exclusion is guaranteed, and progress is also guaranteed. However, a process may be starved, although with decreasing probability, as its number of attempts increases.

5.3.3 Hardware support for mutual exclusion

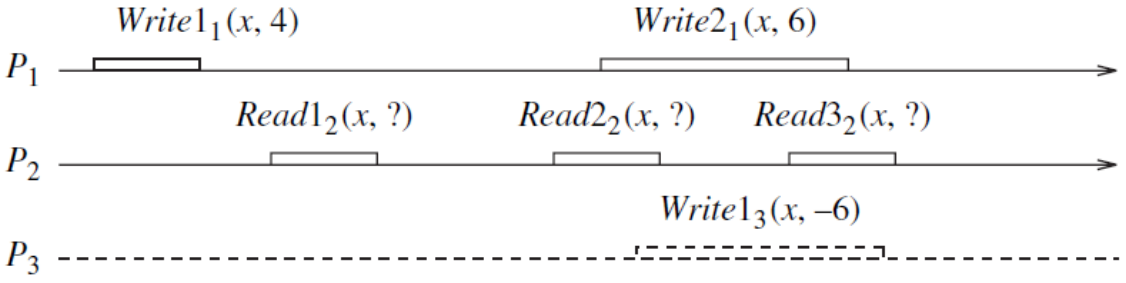
Hardware support can allow for special instructions that perform two or more operations atomically. Two such instructions, Test&Set and Swap , are defined and implemented as shown in Algorithm. The atomic execution of two actions (a Read and a Write operation) can greatly simplify a mutual exclusion algorithm, as seen from the mutual exclusion code in Algorithm, respectively. Algorithm can lead to starvation. Algorithm is enhanced to guarantee bounded waiting by using a “round-robin” policy to selectively grant permission when releasing the critical section.

```
(shared variables among the processes accessing each of the different object
types)
register: Reg ⟵ initial value;                    // shared register initialized
(local variables)
integer: old ⟵ initial value;                      // value to be returned
(1)   Test&Set(Reg) returns value:
(1a)  old ⟵ Reg;
(1b)  Reg ⟵ 1;
(1c)  return(old).
(2)   Swap(Reg, new) returns value:
(2a)  old ⟵ Reg;
(2b)  Reg ⟵ new;
(2c)  return(old).
```

5.3.4 Register hierarchy and wait-free simulations

Observe from our analysis of DSM consistency models that an underlying assumption was that any memory access takes a finite time interval, and the operation, whether a *Read* or

Write, takes effect at some point during



this time duration. In the face of concurrent accesses to a memory location, hereafter called a *register*, we cannot predict the outcome. In particular, in the face of a concurrent *Read* and *Write* operation, the value returned by the *Read* is unpredictable. This observation is true even for a simpler multiprocessor memory, without the context of a DSM. This observation led to the research area that tried to define the properties of access orderings for the most elementary memory unit. The access orderings depend on the properties of the register. An implicit assumption is that of the availability of global time. This is a reasonable assumption because we are studying access to a single register. Whether that register value is replicated in the system or not is a lower detail that is not relevant to the level of abstraction of this analysis.

In keeping with the semantics of the *Read* and *Write* operations, the following register types have been identified by Lamport to specify the value returned to a *Read* in the face of a concurrent *Write* operation. For the time being, we assume that there is a single reader process and a single writer process.

• **Safe register** A *Read* operation that does not overlap with a *Write* operation returns the most recent value written to that register. A *Read* operation that does overlap with a *Write* operation returns *any one* of the values that the register could possibly contain at any time.

Consider the example of Figure, which shows several operations on an integer-valued register. We consider two cases, without and with the *Write* by P3:

– No*Write* by P3 If the register is *safe*, *Read*1*2* must return the value 4, whereas *Read*2*2* and *Read*3*2* can return any possible integer (up to MAXINT) because these operations overlap with a *Write*, and the value returned is therefore ambiguous.

– *Write* by P3 Same as for the "no *Write*" case. If multiple writers are allowed, or if *Write* operations are allowed to be pipelined, then what defines the most recent value of the register in the face of concurrent *Write* operations becomes complicated. We explicitly disallow pipelining in this model and analysis. In the face of *Write* operations from different processors that overlap in time, the notion of a *serialization point* is defined. Observe that each *Write* or *Read* operation has a finite duration between its invocation and its response. In this duration, there iseffectively a single time instant at which the operation takes effect. For a *Read* operation, this instant is the one at which the instantaneous value is selected to be returned. For a *Write* operation, this instant is the one at which the value written is first "reflected" in the register. Using this notion of the serialization point, the "most recent" operation is unambiguously defined.

• **Regular register** In addition to being a *safe* register, a *Read* that is concurrent with a *Write* operation returns either the value before the *Write* operation, or the value written by the *Write* operation.

In the example of Figure, we consider the two cases, with and without the *Write* by P3:

– No *Write* by P3 *Read1*2 must return 4, whereas *Read2*2 can return either 4 or 6, and *Read3*2 can also return either 4 or 6.

– *Write* by P3 *Read1*2 must return 4, whereas *Read2*2 can return either 4 or −6 or 6, and *Read3*2 can also return either 4 or −6 or 6.

• **Atomic register** In addition to being a *regular* register, the register is linearizable to a sequential register.

In the example of Figure, we consider the two cases, with and without the *Write* by P3:

– No *Write* by P3 *Read1*2 must return 4, whereas *Read2*2 can return either 4 or 6. If *Read2*2 returns 4, then *Read3*2 can return either 4 or 6, but if *Read2*2 returns 6, then *Read3*2 must also return 6.

– *Write* by P3 *Read1*2 must return 4, whereas *Read2*2 can return either4 or −6 or 6, depending on the serialization points of the operations. 1. If *Read2*2 returns 6 and the serialization point of *Write1*3 precedes the serialization point of *Write2*1, then *Read3*2 must return 6.

2. If *Read2*2 returns 6 and the serialization point of *Write2*1 precedes the serialization point of *Write1*3, then *Read3*2 can return +6 or −6.

3. Cases (3) and (4) where *Read2*2 returns −6 are similar to cases (1) and (2).

The following properties, summarized in Table 12.2, characterize registers:

• whether the register is single-valued (boolean) or multi-valued

• whether the register is a single-reader (SR) or multi-reader (MR) register

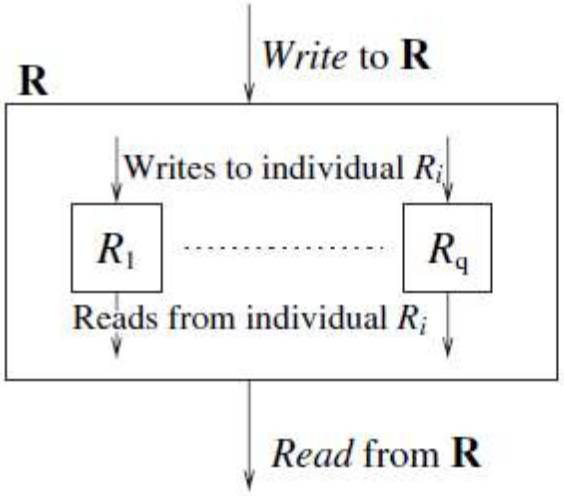
• whether the register is a single-writer (SW) or multi-writer (MW) register

• whether the register is *safe*, *regular*, or *atomic*

The above characteristics lead to a hierarchy of 24 register types, with the most elementary being the boolean SRSW safe register and the most complex being the multi-valued MRMW atomic register. A study of *register construction* deals with designing the more complex registers using simpler registers. Such constructions allow us to construct

Table 12.2 Classification of registers by type, value, writing access, and reading access. The strength of the register increases down each column.

Type	Value	Writing	Reading
safe	binary	single-writer	single-reader
regular	integer	multi-writer	multi-reader
atomic			



any register type from the most elementary register – the boolean SRSW safe register. We will study such constructions by assuming the following convention: R1 Rq are q registers that are used to construct a stronger register R, as shown in Figure. We assume n processes exist; note that for various constructions, q may be different from n. Although the traditional memory architecture, based on serialized access via memory ports to a memory location, does not require such an elaborate classification, the bigger picture needs to be kept in mind. In addition to illustrating algorithmic design techniques, this study paves the way for accommodating newer technologies such as quantum computing and DNA computing for constructing system memory.

5.4 Wait-free atomic snapshots of shared objects

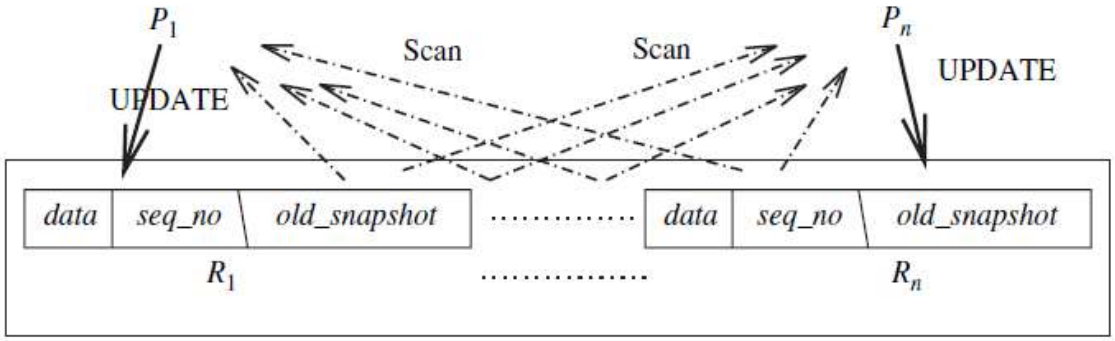
Observing the global state of a distributed system is a fundamental problem. For message-passing systems, we have studied how to record global snapshots which represent an instantaneous possible global state that could have occurred in the execution. The snapshot algorithms used message-passing of control messages, and were inherently inhibition-free, although some variants that use fewer control messages do require inhibition. In this section,

we examine the counterpart of the global snapshot problem in a shared-memory system, where only *Read* and *Write* primitives can be used. The problem can be modeled as follows. Given a set of SWMR atomic registers R_1...R_n, where R_i can be written only by P_i and can be read by all processes, and which together form a compound high-level object, devise a *wait-free* algorithm to observe the state of the object at some instant in time. The following actions are allowed on this high-level object, as also illustrated in Figure:

• *Scan*$_i$: This action invoked by P_i returns the atomic snapshot that is an instantaneous view of the object (R_1)...(R_n) at some instant between the invocation and termination of the *Scan*.

• *Update*$_i$_val_: This action invoked by Pi writes the data val to register R_i.

Clearly, any kind of locking mechanism is unacceptable because it is not wait-free. Consider the following attempt at a wait-free solution. The format of each register R_i is assumed to be the tuple: (data, seq_no) in order to uniquely identify each *Write* operation to the register. A scanner would repeatedly scan the high-level object until two consecutive scans, called *double-collect* in the shared memory context, returned identical content. This principle of "double-collect" has been encountered in multiple contexts, such in two phase deadlock detection and two-phase termination detection algorithms, and essentially embodies the two-phase observation rule. However, this solution in not wait-free because between the two observations of each double-collect, an *Update* by another process can prevent the *Scan* from being successful. A wait-free solution is given in Algorithm. Process Pi can write to its RSW register R_i and can read all registers R_1' R_n. To design a wait free solution, it needs to be ensured that a scanner is not indefinitely prevented from getting identical scans in the double-collect, by some writer process periodically making updates. The problem arises because of the imbalance in the roles of the scanner and updater – the updater is inherently more powerful in that it can prevent all scanners from being successful. One elegant solution therefore neutralizes the unfair advantage of the updaters by forcing



Snapshot object composed of *n* MRSW atomic registers

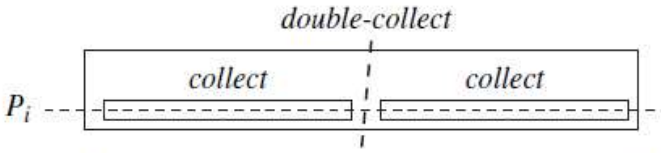
the updaters to follow the same rules as the scanner. Namely, the updaters also have to perform a double-collect, and only after performing a double collect can an update write the value it needs to. Additionally, an updater also writes the snapshot it collected in the register, along with the new value of the data item. Now, if a scanner detects that an updater has made an update after the scanner initiated its *Scan*, then the scanner can simply "borrow" the snapshot recorded by the updater in its register. The updater helps the scanner to obtain a

consistent value. This is the principle of "helping" that is often used in designing wait-free solutions for various problems.

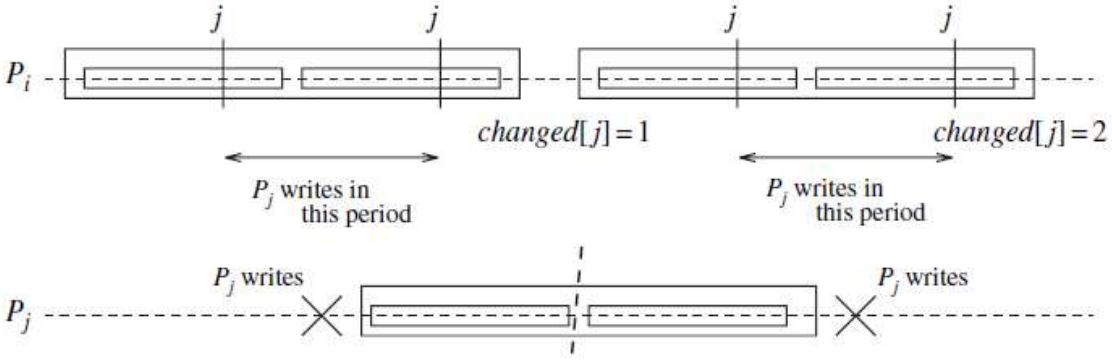
```
(shared variables)
MRSW atomic register of type ⟨data, seq_no, old_snapshot⟩, where
data, seq_no are of type integer, and old_snapshot is array [1...n] of
integer: R_1...R_n;
(local variables)
integer: changed[1...n];
type ⟨data, seq_no, old_snapshot⟩: v1[1...n], v2[1...n], v[1...n];
(1)  Update_i(x)
(1a) v[1...n] ⟵ Scan_i;
(1b) R_i ⟵ (x, R_i.seq_no + 1, v[1...n]).

(2)  Scan_i
(2a) for count = 1 to n do
(2b)       changed[count] ⟵ 0;
(2c) while true do
(2d)       v1[1...n] ⟵ collect();
(2e)       v2[1...n] ⟵ collect();
(2f)       if (∀k, 1 ≤ k ≤ n)(v1[k].seq_no = v2[k].seq_no) then
(2g)             return(v2[1].data, ..., v2[n].data);
(2h)       else
(2i)             for k = 1 to n do
(2j)                   if v1[k].seq_no ≠ v2[k].seq_no then
(2k)                         changed[k] ⟵ changed[k] + 1;
(2l)                         if changed[k] = 2 then
(2m)                               return(v2[k].old_snapshot).
```

A scanner detects that an updater has made an update after the scanner initiated its *Scan*, by using the local array changed. This array is reset to 0 when the *Scan* is invoked. Location changed(k) is incremented (line 2k) if the *Scan* procedure detects (line 2j) that process P_k has changed its data and seq_no (and implicitly the old_snapshot) fields in Rk. Based on the

value



(a) *double-collect* sees identical values in both *collects*



(b) P_j's *double-collect* nested within P_i's SCAN. The *double-collect* is successful, or P_j borrowed snapshot from P_k's *double-collect* nested within P_j's *Scan*. And so on recursively, up to *n* times.

of changed(k), different inferences can be made, as now explained with the help of Figure:

• If changed(k)= 2 (line 2l), then two updates (line 1b) were made by P_k after Pi began its *Scan*. Between the first and the second update, the *Scan* preceding the second update must have completed successfully, and the scanned value was recorded in the old_snapshot field. This old snapshot can be safely borrowed by the scanner Pi (line 2m) because it was recorded after P_k finished its first *double-collect*, and hence after the scanner Pi initiated its *Scan*.

• However, if changed(k) = 1, it cannot be inferred that the old_snapshot recorded by P_k was taken after Pi's *Scan* began. When P_k does its the value it writes in *old_snapshot* is only the result of a double-scan that preceded the "write" and may be a value that existed before Pi's *Scan* began. There are two cases by which a snapshot can be captured, as illustrated using Figure :

1. A scanner can collect a snapshot (line 2g) if the double-collect (lines 2d–2e) returns identical views (line 2f). The returned snapshot represents an instantaneous state that existed at all times between the end of the first *collect* (line 2d) and the start of the second *collect* (line 2e). Otherwise the scanner returns a borrowed snapshot (line 2m) from P_k if P_k has been noticed to have made two updates (lines 2l) and therefore P_k has made a *Scan* embedded inside Pi's *Scan*. This borrowed snapshot itself (i) may have been obtained directly via a *double-collect*, or (ii) indirectly been borrowed from another process (line 2l). In case (i), it represents an instantaneous state/ in the duration of the *double-collect*. In case (ii), a recursive argument can be applied. Observe that there are n processes, so the recursive argument can hold at most n−1 times. The nth time, a *double-collect* must have been successful. Note that between the two *double-collect*s of Pi that are shown, there may be up to (n−2) other

unsuccessful *double-collect*s of Pi. Each of these (n−2) other *double-collect*s corresponds to some P_k, k '= (i, j), having "changed" once.

The linearization of the *Scan* and *Update* operations follows in a straightforward manner. For example, non-overlapping operations get linearized in the order of their occurrence. An operation by Pi that borrows a snapshot from P_k gets linearized after P_k.

Complexity

The local space complexity is $O(n^2)$ integers. The shared space is $O(n^2)$ corresponding to each of the n registers of size O(n)each. The time complexity is $O(n^2)$. This is because the main *Scan* loop has a complexity of O(n) and the loop may be executed at most n times – the nth time, at least one process P_k must have caused Pi's local changed_k' to reach a value of two, triggering an end to the loop (lines 2k–2l).

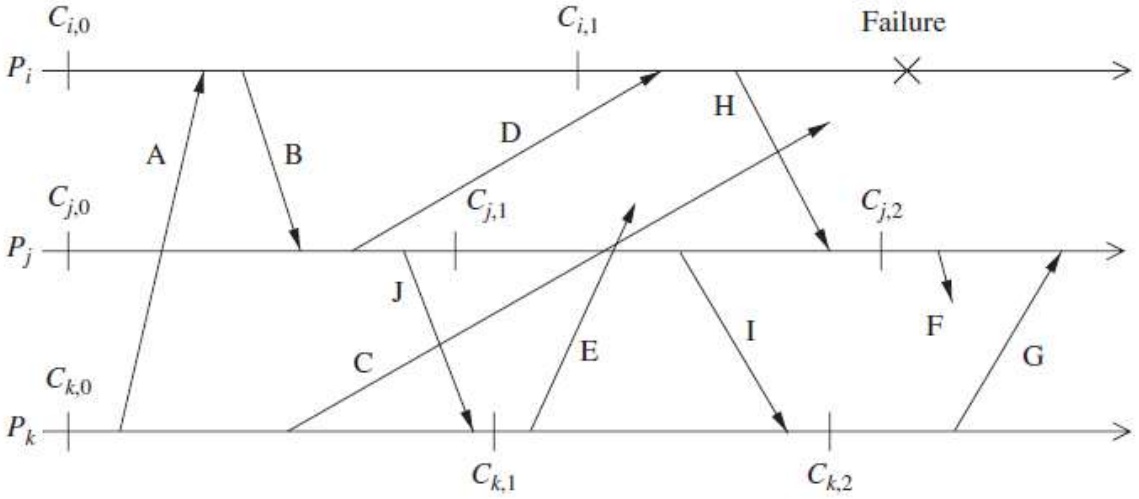
5.5 Issues in failure recovery

In a failure recovery, we must not only restore the system to a consistent state, but also appropriately handle messages that are left in an abnormal state due to the failure and recovery.

We now describe the issues involved in a failure recovery with the help of a distributed computation shown in Figure. The computation comprises of three processes P_i, P_j , and P_k, connected through a communication network. The processes communicate solely by exchanging messages over fault-free, FIFO communication channels. Processes Pi, Pj , and Pk have taken checkpoints {Ci_0, Ci_1}, {Cj_0, Cj_1, Cj_2}, and {Ck_0, Ck_1}, respectively, and these processes have exchanged messages A to J as shown in Figure.

Suppose process Pi fails at the instance indicated in the figure. All the contents of the volatile memory of Pi are lost and, after Pi has recovered from the failure, the system needs to be restored to a consistent global state from where the processes can resume their execution. Process Pi's state is restored to a valid state by rolling it back to its most recent checkpoint Ci_1. To restore the system to a consistent state, the process Pj rolls back to checkpoint Cj_1 because the rollback of process Pi to checkpoint Ci_1 created an orphan message H (the receive event of H is recorded at process Pj while the send event of H has been undone at process Pi). Note that process Pj does not roll back to checkpoint Cj_2 but to checkpoint Cj_1, because rolling back to checkpoint Cj_2 does not eliminate the orphan message H. Even this resulting state is not a consistent global state, as an orphan message I is created due to the roll back of process Pj to checkpoint Cj_1. To eliminate this orphan message, process

Pk rolls back to checkpoint Ck_1. The



restored global state $\{C_{i,1}, C_{j\_1}, C_{k\_1}\}$ is a consistent state as it is free from orphan messages. Although the system state has been restored to a consistent state, several messages are left in an erroneous state which must be handled correctly.

Messages A, B, D, G, H, I, and J had been received at the points indicated in the figure and messages C, E, and F were in transit when the failure occurred. Restoration of system state to checkpoints $\{C_{i,1}, C_{j,1}, C_{k,1}\}$ automatically handles messages A, B, and J because the send and receive events of messages A, B, and J have been recorded, and both the events for G, H, and I have been completely undone. These messages cause no problem and we call messages A, B, and J normal messages and messages G, H, and I vanished messages.

Messages C, D, E, and F are potentially problematic. Message C is in transit during the failure and it is a delayed message. The delayed message C has several possibilities: C might arrive at process Pi before it recovers, it might arrive while Pi is recovering, or it might arrive after Pi has completed recovery. Each of these cases must be dealt with correctly. Message D is a lost message since the send event for D is recorded in the restored state for process P_j , but the receive event has been undone at process Pi. Process P_j will not resend D without an additional mechanism, since the send D at P_j occurred before the checkpoint and the communication system successfully delivered D.

Messages E and F are delayed orphan messages and pose perhaps the most serious problem of all the messages. When messages E and F arrive at their respective destinations, they must be discarded since their send events have been undone. Processes, after resuming execution from their checkpoints, will generate both of these messages, and recovery techniques must be able to distinguish between messages like C and those like E and F.

Lost messages like D can be handled by having processes keep a message log of all the sent messages. So when a process restores to a checkpoint, it replays the messages from its log to handle the lost message problem. However, message logging and message replaying during recovery can result in duplicate messages. In the example shown in Figure, when process P_j replays messages from its log, it will regenerate message J. Process P_k, which has already received message J, will receive it again, thereby causing inconsistency in the system state. Therefore, these duplicate messages must be handled properly.

Overlapping failures further complicate the recovery process. A process P_j that begins rollback/recovery in response to the failure of a process P can itself fail and develop amnesia with respect process P_i's failure; that is, process P_j can act in a fashion that exhibits ignorance of process P_i's failure. If overlapping failures are to be tolerated, a mechanism must be introduced to deal with amnesia and the resulting inconsistencies.

5.6 Checkpoint-based recovery

In the checkpoint-based recovery approach, the state of each process and the communication channel is check pointed frequently so that, upon a failure, the system can be restored to a globally consistent set of checkpoints. It does not rely on the PWD assumption, and so does not need to detect, log, or replay non-deterministic events. Checkpoint-based protocols are therefore less restrictive and simpler to implement than log-based rollback recovery.

However, checkpoint-based rollback recovery does not guarantee that prefailure execution can be deterministically regenerated after a rollback. Therefore, checkpoint-based rollback recovery may not be suitable for applications that require frequent interactions with the outside world. Checkpoint-based rollback-recovery techniques can be classified into three categories: *uncoordinated checkpointing*, *coordinated checkpointing*, and *communication-induced checkpointing.*

5.6.1 Uncoordinated checkpointing

In uncoordinated checkpointing, each process has autonomy in deciding when to take checkpoints. This eliminates the synchronization overhead as there is no need for coordination between processes and it allows processes to take checkpoints when it is most convenient or efficient. The main advantage is the lower runtime overhead during normal execution, because no coordination among processes is necessary. Autonomy in taking checkpoints also allows each process to select appropriate checkpoints positions. However, uncoordinated checkpointing has several shortcomings.

First, there is the possibility of the domino effect during a recovery, which may cause the loss of a large amount of useful work.

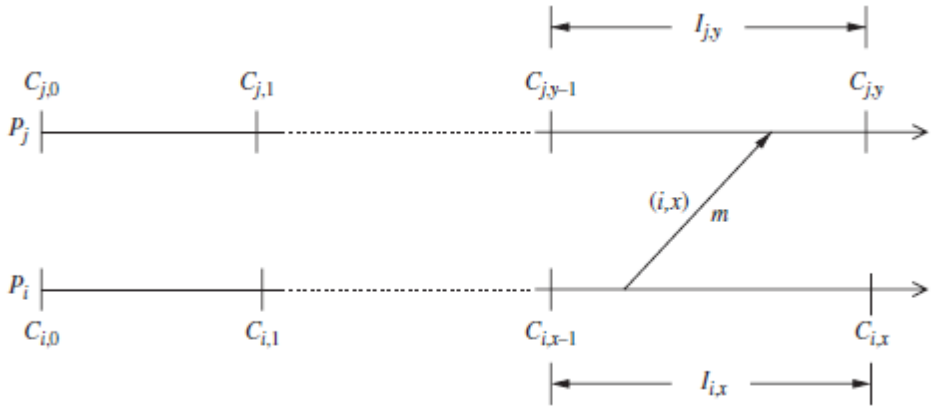
Second, recovery from a failure is slow because processes need to iterate to find a consistent set of checkpoints. Since no coordination is done at the time the checkpoint is taken, checkpoints taken by a process may be *useless* checkpoints. (A useless checkpoint is never a part of any global consistent state.) Useless checkpoints are undesirable because they incur overhead and do not contribute to advancing the recovery line.

Third, uncoordinated checkpointing forces each process to maintain multiple checkpoints, and to periodically invoke a garbage collection algorithm to reclaim the checkpoints that are no longer required.

Fourth, it is not suitable for applications with frequent output commits because these require global coordination to compute the recovery line, negating much of the advantage of autonomy.

As each process takes checkpoints independently, we need to determine a consistent global checkpoint to rollback to, when a failure occurs. In order to determine a consistent global checkpoint during recovery, the processes record the dependencies among their checkpoints

caused by message exchange



during failure-free operation. The following direct dependency tracking technique is commonly used in uncoordinated checkpointing.

Let $C_{i,x}$ be the x^{th} checkpoint of process Pi, where i is the process i.d. and x is the checkpoint index (we assume each process Pi starts its execution with an initial checkpoint $C_{i,0}$). Let $I_{i,x}$ denote the *checkpoint interval* or simply *interval* between checkpoints $C_{i,x-1}$ and $C_{i,x.}$ Consider the example shown in Figure. When process P_i at interval $I_{i,x}$ sends a message m to P_j , it piggybacks the pair (i, x) on m. When P_j receives m during interval $I_{j,y}$, it records the dependency from $I_{i,x}$ to $I_{j,y}$, which is later saved onto stable storage when P_j takes checkpoint $C_{j,y}$. When a failure occurs, the recovering process initiates rollback by broadcasting a *dependency request* message to collect all the dependency information maintained by each process. When a process receives this message, it stops its execution and replies with the dependency information saved on the stable storage as well as with the dependency information, if any, which is associated with its current state. The initiator then calculates the recovery line based on the global dependency information and broadcasts a *rollback request* message containing the recovery line. Upon receiving this message, a process whose current state belongs to the recovery line simply resumes execution; otherwise, it rolls back to an earlier checkpoint as indicated by the recovery line.

5.6.2 Coordinated checkpointing

In coordinated checkpointing, processes orchestrate their checkpointing activities so that all local checkpoints form a consistent global state. Coordinated checkpointing simplifies recovery and is not susceptible to the domino effect, since every process always restarts from its most recent checkpoint. Also, coordinated checkpointing requires each process to maintain only one checkpoint on the stable storage, reducing the storage overhead and eliminating the need for garbage collection. The main disadvantage of this method is that large latency is involved in committing output, as a global checkpoint is needed before a message is sent to the OWP. Also, delays and overhead are involved everytime a new global checkpoint is taken. If perfectly synchronized clocks were available at processes, the following simple method can be used for checkpointing: all processes agree at what instants of time they will take checkpoints, and the clocks at processes trigger the local checkpointing actions at all processes. Since perfectly synchronized clocks are not available, the following approaches

are used to guarantee checkpoint consistency: either the sending of messages is blocked for the duration of the protocol, or checkpoint indices are piggybacked to avoid blocking.

Blocking coordinated checkpointing

A straightforward approach to coordinated checkpointing is to block communications while the checkpointing protocol executes. After a process takes a local checkpoint, to prevent orphan messages, it remains blocked until the entire checkpointing activity is complete. The coordinator takes a checkpoint and broadcasts a request message to all processes, asking them to take a checkpoint. When a process receives this message, it stops its execution, flushes all the communication channels, takes a *tentative* checkpoint, and sends an acknowledgment message back to the coordinator. After the coordinator receives acknowledgments from all processes, it broadcasts a commit message that completes the two-phase checkpointing protocol. After receiving the commit message, a process removes the old permanent checkpoint and atomically makes the *tentative* checkpoint permanent and then resumes its execution and exchange of messages with other processes. A problem with this approach is that the computation is blocked during the checkpointing and therefore, non-blocking checkpointing schemes are preferable.

Non-blocking checkpoint coordination

In this approach the processes need not stop their execution while taking checkpoints. A fundamental problem in coordinated checkpointing is to prevent a process from receiving application messages that could make the checkpoint inconsistent. Consider the example in Figure: message *m* is sent by *P*0 *after* receiving a checkpoint request from the checkpoint coordinator. Assume *m* reaches *P*1 *before* the checkpoint request. This situation results in an inconsistent checkpoint since checkpoint $c_{1,x}$ shows the receipt of message *m* from P_0, while checkpoint $c_{0,x}$ does not show *m* being sent from P_0. If channels are FIFO, this problem can be avoided by preceding the first post-checkpoint message on each channel by a checkpoint request, forcing each process to take a checkpoint before receiving the first post-checkpoint message, as illustrated in Figure. An example of a non-blocking checkpoint coordination protocol using this idea is the snapshot algorithm of Chandy and Lamport in which *markers* play the role of the checkpoint request messages. In this algorithm, the initiator takes a checkpoint and sends a marker (a checkpoint request) on all outgoing channels. Each process takes a checkpoint upon receiving the first marker and sends the marker on all outgoing channels before sending any application message. The protocol works assuming the channels are reliable and FIFO.

If the channels are non-FIFO, the following two approaches can be used: first, the marker can be piggybacked on every post-checkpoint message. When a process receives an application message with a marker, it treats it as if it has received a marker message, followed by the application message. Alternatively, checkpoint indices can serve the same role as markers, where a checkpoint is triggered when the receiver's local checkpoint index is lower than the piggybacked checkpoint index.

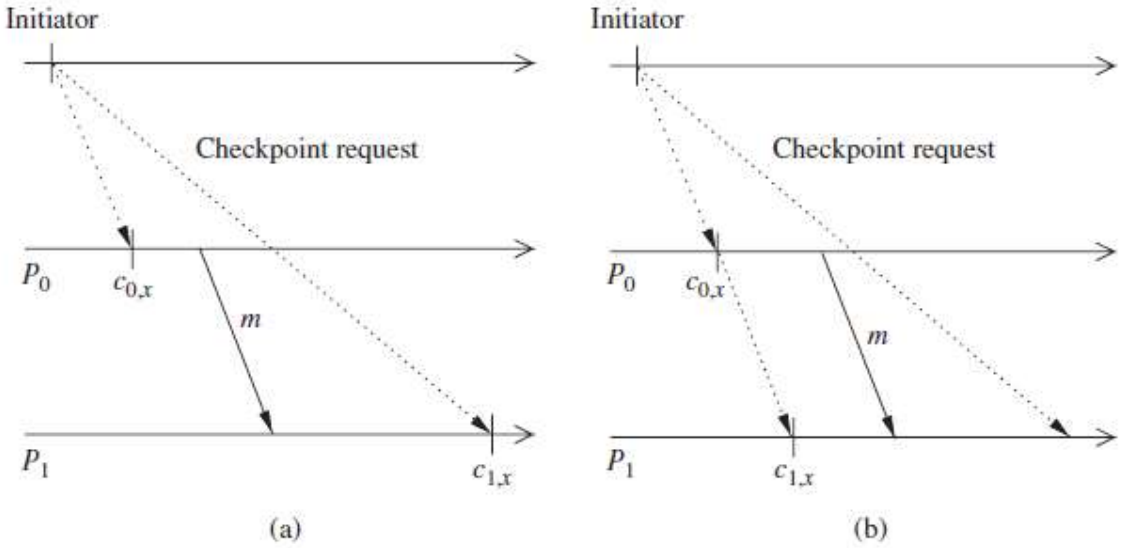
Coordinated checkpointing requires all processes to participate in every checkpoint. This requirement generates valid concerns about its scalability. It is desirable to reduce the number

of processes involved in a coordinated checkpointing session. This can be done since only those processes that have communicated with the checkpoint initiator either directly or indirectly since the last checkpoint need to take new checkpoints. A two-phase protocol by Koo and Toueg achieves minimal checkpoint coordination.

5.6.3 Impossibility of min-process non-blocking checkpointing

A min-process, non-blocking checkpointing algorithm is one that forces only a minimum number of processes to take a new checkpoint, and at the same time it does not force any process to suspend its computation. Clearly, such checkpointing algorithms will be very attractive. Cao and Singhal showed that it is impossible to design a min-process, non-blocking checkpointing algorithm.

Of course, the following type of min-process checkpointing algorithms are possible. The algorithm consists of two phases. During the first phase, the



checkpoint initiator identifies all processes with which it has communicated since the last checkpoint and sends them a request. Upon receiving the request, each process in turn identifies all processes it has communicated with since the last checkpoint and sends them a request, and so on, until no more processes can be identified. During the second phase, all processes identified in the first phase take a checkpoint. The result is a consistent checkpoint that involves only the participating processes. In this protocol, after a process takes a checkpoint, it cannot send any message until the second phase terminates successfully, although receiving a message after the checkpoint has been taken is allowable.

Based on a concept called "Z-dependency," Cao and Singhal proved that there does not exist a non-blocking algorithm that will allow a minimum number of processes to take their checkpoints. Here we give only a sketch of the proof and readers are referred to the original source for a detailed proof.

Z-dependency is defined as follows: if a process P_p sends a message to process P_q during its i^{th} checkpoint interval and process P_q receives the message during its j^{th} checkpoint interval, then Pq Z-depends on Pp during Pp's i^{th} checkpoint interval and Pq's j^{th} checkpoint interval, denoted by Pp $\rightarrow$i jPq. If Pp $\rightarrow$i, j Pq and Pq $\rightarrow$ j kPr , then Pr transitively Z-depends

depends on Pp during Pr 's kth checkpoint interval and Pp's ith checkpoint interval, and this is denoted as Pp ∗ →i kPr . A min process algorithm is one that satisfies the following condition: when a process Pp initiates a new checkpoint and takes checkpoint Cp_i, a process Pq takes a checkpoint Cq_jassociated with Cp_i if and only if Pq ∗ →j−1 i−1Pp. In a min-process non-blocking algorithm, process Pp initiates a new checkpoint and takes a checkpoint Cp_i and if a process Pr sends a message m to Pq after it takes a new checkpoint associated with Cp_i, then Pq takes a checkpoint Cq_i before processing m if and only if Pq ∗ → j−1 i−1Pp. According to the min-process definition, Pq takes checkpoint Cq_j if and only if Pq ∗ →j−1 i−1Pp, but Pq should take Cq_i before processing m. If it takes Cq_j after processing m, m becomes an orphan. Therefore, when a process receives a message m, it must know if the initiator of a new checkpoint transitively Z-depends on it during the previous checkpoint interval. But it has been proved that there is not enough information at the receiver of a message to decide whether the initiator of a new checkpoint transitively Z-depends on the receiver. Therefore, no min-process, non-blocking algorithm exists.

5.6.4 Communication-induced checkpointing

Communication-induced checkpointing is another way to avoid the domino effect, while allowing processes to take some of their checkpoints independently Processes may be forced to take additional checkpoints (over and above their autonomous checkpoints), and thus process independence is constrained to guarantee the eventual progress of the recovery line.

Communication-induced checkpointing reduces or completely eliminates the useless checkpoints. In communication-induced checkpointing, processes take two types of checkpoints, namely, autonomous and forced checkpoints. The checkpoints that a process takes independently are called *local* checkpoints, while those that a process is forced to take are called *forced* checkpoints. Communication-induced checkpointing piggybacks protocol-related information on each application message. The receiver of each application message uses the piggybacked information to determine if it has to take a forced checkpoint to advance the global recovery line. The forced checkpoint must be taken before the application may process the contents of the message, possibly incurring some latency and overhead. It is therefore desirable in these systems to minimize the number of forced checkpoints. In contrast with coordinated checkpointing, no special coordination messages are exchanged.
There are two types of communication-induced checkpointing: modelbased checkpointing and index-based checkpointing. In *model-based checkpointing*, the system maintains checkpoints and communication structures that prevent the domino effect or achieve some even stronger properties. In *index-based checkpointing*, the system uses an indexing scheme for the local and forced checkpoints, such that the checkpoints of the same index at all processes form a consistent state.

Model-based checkpointing

Model-based checkpointing prevents patterns of communications and checkpoints that could result in inconsistent states among the existing checkpoints. A process detects the potential for inconsistent checkpoints and independently forces local checkpoints to prevent the formation of undesirable patterns. A forced checkpoint is generally used to prevent the

undesirable patterns from occurring. No control messages are exchanged among the processes during normal operation. All information necessary to execute the protocol is piggybacked on application messages. The decision to take a forced checkpoint is done locally using the information available. There are several domino-effect-free checkpoint and communication models.

The *MRS* (mark, send, and receive) model of Russell avoids the domino effect by ensuring that within every checkpoint interval all message receiving events precede all message-sending events. This model can be maintained by taking an additional checkpoint before every message-receiving event that is not separated from its previous message-sending event by a checkpoint. Another way to prevent the domino effect by avoiding rollback propagation completely is by taking a checkpoint immediately after every message-sending event. Recent work has focused on ensuring that every checkpoint can belong to a consistent global checkpoint and therefore is not useless.

Index-based checkpointing

Index-based communication-induced checkpointing assigns monotonically increasing indexes to checkpoints, such that the checkpoints having the same index at different processes form a consistent state. Inconsistency between checkpoints of the same index can be avoided in a lazy fashion if indexes are piggybacked on application messages to help receivers decide when they should take a forced a checkpoint. For instance, the protocol by Briatico *et al.* forces a process to take a checkpoint upon receiving a message with a piggybacked index greater than the local index. More sophisticated protocols piggyback more information on application messages to minimize the number of forced checkpoints.

5.7 Authentication in distributed systems

A fundamental concern in building a secure distributed system is the authentication of local and remote entities in the system. In a distributed system, the hosts communicate by sending and receiving messages over the network. Various resources (such as files and printers) distributed among the hosts are shared across the network in the form of network services provided by servers. The entities in a distributed system, such as users, clients, servers, and processes, are collectively referred to as principals. A distributed system is susceptible to a variety of threats mounted by intruders as well as legitimate users of the system.

In an environment where a principal can impersonate another principal, principals must adopt a mutually suspicious attitude toward one another and authentication becomes an important requirement. Authentication is a process by which one principal verifies the identity of another principal. For example, in a client–server system, the server may need to authenticate the client. Likewise, the client may want to authenticate the server so that it is assured that it is talking to the right entity. Authentication is needed for both authorization and accounting functions. In one-way authentication, only one principal verifies the identity of the other principal, while in mutual authentication both communicating principals verify each other's identity. A user gains access to a distributed system by logging on to a host in the system. In an open access environment where hosts are scattered across unrestricted areas, a host can be arbitrarily compromised, necessitating mutual authentication between the user and host. In a

distributed system, authentication is carried out using a protocol involving message exchanges and these protocols are termed *authentication protocols*.

5.7.1 Background and definitions

In simple terms, authentication is identification plus verification. *Identification* is the procedure whereby an entity claims a certain identity, while *verification* is the procedure whereby that claim is checked. Authentication is a process of verifying that the principal's identity is as claimed. The *correctness* of authentication relies heavily on the verification procedure employed.

A successful identity authentication results in a belief held by the authenticating principal (the *verifier*) that the authenticated principal (the *claimant*) possesses the claimed identity. The other types of authentication include message origin authentication and message content authentication. In this chapter, we restrict our attention to identity authentication only. Authentication in distributed systems is carried out using protocols. A protocol is a precisely defined sequence of communication and computation steps. A communication step transfers messages from one principal (the sender) to another (the receiver), while a computation step updates a principal's internal state. Two distinct states can be identified upon the termination of the protocol: one signifying successful authentication and the other failure.

Although the goal of any authentication is to verify the claimed identity of a principal, specific success and failure states are highly protocol dependent. For example, the success of an authentication during the connection establishment phase of a communication protocol is usually indicated by the distribution of a fresh session key between two mutually authenticated peer processes. On the other hand, in a user login authentication, success usually results in the creation of a login process on behalf of the user.

5.7.2 Basis of authentication

Authentication generally is based on the possession of some secret information, like password, known only to the entities participating in the authentication. When an entity wants to authenticate another entity, the former will verify if the latter possesses the knowledge of the secret. If the entity demonstrates the knowledge of the right secret information, the authentication succeeds, else authentication fails. Examples of secret information for the purpose of authentication include the following: something known (e.g., a shared key), something possessed (e.g., a smartcard), or something inherent (e.g., biometrics). However, the verification process should not allow an attacker to reuse an authentication exchange to impersonate an entity. The verification process must provide the verifier with enough confidence that an attacker is not trying to impersonate an entity.

5.7.3 Types of principals

In a distributed system, the entities that require identification are hosts, users, and processes. They thus are the principals involved in an authentication.

• **Hosts** These are addressable entities at the network level. A host is usually identified by its name (for example, a fully qualified domain name) or its network address (for example, an IP address).

• **Users** These entities are ultimately responsible for all system activities. Users initiate and are accountable for all system activities. Most access control and accounting functions are based on users. Typical users include humans, as well as accounts maintained in the user database. Users are considered to be outside the system boundary.
• **Processes** The system creates processes within the system boundary to represent users. A process requests and consumes resources on the behalf of its user.
Processes fall into two classes: client and server. Client processes are consumers who obtain services from server processes, who are service providers. A particular process can act as both a client and a server.

5.7.4 A simple classification of authentication protocols

Authentication protocols can be categorized based on the following criteria : type of cryptography (symmetric vs. asymmetric), reciprocity of authentication (mutual vs. one-way), key exchange, real-time involvement of a third party (on-line vs. off-line), nature of trust required from a third party, nature of security guarantees, and storage of secrets.
In this chapter, we classify authentication protocols primarily based on the cryptographic technique used. There are two basic types of cryptographic techniques: symmetric ("private key") and asymmetric ("public key"). Symmetric cryptography uses a single private key to both encrypt and decrypt data. Any party that has the key can use it to encrypt and decrypt data. Symmetric cryptography algorithms are typically fast and are suitable for processing large streams of data. Asymmetric cryptography, also called public-key cryptography, uses a secret key that must be kept from unauthorized users and a public key that is made public. Both the public key and the private key are mathematically linked: data encrypted with the public key can be decrypted only by the corresponding private key, and data signed with the private key can only be verified with the corresponding public key. Both keys are unique to a communication session.

5.7.5 Design principles for cryptographic protocols

Abadi and Needham set out a set of principles to denote prudent engineering practices for cryptographic protocols design. They are not meant to apply to every protocol in every instance, but they do provide rules of thumb that should be considered when designing a cryptographic protocol.
We next present these principles and briefly comment on them.
• **Principle 1** Every message should say what it means: the interpretation of the message should depend only on its content. It should be possible to write down a straightforward English sentence describing the content – though if there is a suitable formalism available, which is good, too.
• **Principle 2** The conditions for a message to be acted upon should be clearly set out so that someone reviewing the design may see whether they are acceptable or not.
• **Principle 3** If the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal's name explicitly in the message.
• **Principle 4** Be clear as to why encryption is being done. Encryption is not wholly cheap, and not asking precisely why it is being done can lead to redundancy. Encryption is not synonymous with security, and its improper use can lead to errors.

• **Principle 5** When a principal signs material that has already been encrypted, it should not be inferred that the principal knows the content of the message. On the other hand, it is proper to infer that the principal that signs a message and then encrypts it for privacy knows the content of the message.
• **Principle 6** Be clear about what properties you are assuming about nonces. What may do for ensuring temporal succession may not do for ensuring association – and perhaps association is best established by other means.
• **Principle 7** The use of a predictable quantity (such as the value of a counter) can serve in guaranteeing newness, through a challenge–response exchange. But if a predictable quantity is to be effective, it should be protected so that an intruder cannot simulate a challenge and later replay a response.
• **Principle 8** If timestamps are used as freshness guarantees by reference to absolute time, then the difference between local clocks at various machines must be much less than the allowable age of a message deemed to be valid. Furthermore, the time maintenance mechanism everywhere becomes part of the trusted computing base.
• **Principle 9** A key may have been used recently, for example, to encrypt a nonce, yet be quite old, and possibly compromised. Recent use does not make the key look any better than it would otherwise.
• **Principle 10** If an encoding is used to present the meaning of a message, then it should be possible to tell which encoding is being used. In the common case where the encoding is protocol dependent, it should be possible to deduce that the message belongs to this protocol, and in fact to a particular run of the protocol, and to know its number in the protocol.
• **Principle 11** The protocol designer should know which trust relations his protocol depends on, and why the dependence is necessary. The reasons for particular trust relations being acceptable should be explicit though they will be founded on judgment and policy rather than on logic.

5.7.6 Protocols based on symmetric cryptosystems

In a symmetric cryptosystem, knowing the shared key lets a principal encrypt and decrypt arbitrary messages. Without such knowledge, a principal cannot create the encrypted version of a message, or decrypt an encrypted message. Hence, authentication protocols can be designed using the following principle:
If a principal can correctly encrypt a message using a key that the verifier believes is known only to a principal with the claimed identity (outside of the verifier), this act constitutes sufficient proof of identity. Thus, the principle embodies the fact that a principal's knowledge is indirectly demonstrated through its ability to encrypt or decrypt.

5.7.6.1 Basic protocol

Using the above principle, we immediately obtain the basic where principal P is authenticating itself to principal Q. "k" denotes a secret key that is shared between only P and Q.

P	:	Create a message $m =$ "I am P."
	:	Compute $m' = \{m, Q\}_k$
$P \rightarrow Q$	:	m, m'
Q	:	verify $\{m, Q\}_k = m'$
	:	if equal then accept; otherwise the authentication fails

In the modified version of the protocol, the principal P wants to authenticate itself to Q. Q generates a nonce and sends this nonce to P. P then encrypts Q, the nonce, and its own identity with the secret key and sends this encrypted message to Q. Q verifies this encrypted message by encrypting its identity, P's identity, and the nonce with the key k. Q authenticates P if the encrypted information equals that sent by P, else the authentication fails.
Replay is foiled by the freshness of nonce n and because n is drawn from a large space. Therefore, it is highly unlikely that the nonce n generated by Q in the current session is the same as one used in a previous session. Thus an attacker cannot use a message of type m_ from a previous session to mount a replay attack. In addition, even if an eavesdropper has monitored all previous authentication conversations between P and Q, it is impossible to produce the message m because it does not know the secret key k. The challenge-and-response step can be repeated any number of times until the desired level of confidence is reached by Q.

Weaknesses
This protocol has scalability problems because each principal must store the secret key for every other principal it would ever want to authenticate. This presents major initialization (the predistribution of secret keys) and storage problems. Moreover, the compromise of one principal can potentially compromise the entire system. Note that this protocol is also vulnerable to known plain text attacks.

5.7.6.2 Wide-mouth frog protocol
The above raised problems can be significantly reduced by postulating a centralized server S. The wide-mouth frog protocol uses a similar approach where a principal A authenticates itself to principal B using a Server S. The protocol works as follows:

$$A \rightarrow S: \quad A, \{T_A, K_{AB}, B\}_{K_{AS}}$$

$$S \rightarrow B: \quad \{T_S, K_{AB}, A\}_{K_{BS}}$$

A decides that it wants to set up communication with B. A sends to S its identity and a packet encrypted with the key, KAS, it shares with S. The packet contains the current timestamp, A's desired communication partner, and a randomly generated key KAB, for communication between A and B. S decrypts the packet to obtain KAB and then forwards this key to B in an encrypted packet that also contains the current timestamp and A's identity. B decrypts this message with the key it shares with S and retrieves the identity of the other party and the key, KAB. Any principal receiving a message with an out-of-date timestamp during this protocol

discards it to prevent replay attacks. This protocol achieve two objectives: first, it securely establishes a secret key between two principals *A* and *B*; and second, A authenticates itself to B with the help of the server S. This is because only the server S could have constructed the message TS_KAB_A_KBS in step 2 only after receiving a message from A in step 1. A weakness of the protocol is that a global clock is required and the protocol will fail if the server S is compromised.

5.7.6.3 A protocol based on an authentication server

Another approach to solve the problem is by using a centralized *authentication server S* that shares a secret key KXS with every principal X in the system. The basic authentication protocol is shown in Algorithm. In the protocol using an authentication server, the principal P sends its identity to Q. Q generates a nonce and sends this nonce to P. P then encrypts P, Q and n with the key KPS and sends this encrypted value x to Q. Q then encrypts P, Q and x with KQS and sends this encrypted value y to authentication server S. Since S knows both the secret keys, it decrypts y with KQS, recovers x, decrypts x with KPS and recovers P, Q and n. Server S then encrypts P, Q and n with key KQS and sends the encrypted value m to Q. Q then computes P, Q and nKQS and verifies if this value is equal to the value received from S. If both values are equal, then authentication succeeds, else it fails.

Thus Q's verification step is preceded by a *key-translation* step by S. Since P and Q do not share a secret key, the authentication server S does the key translation because it shares a secret key with both principals P and Q. Q sends the message (encrypted with KPS that it received from P) to S. S does the key translation by decrypting it with KPS, encrypting P, Q and n with KQS and sending the message encrypted with KQS to Q. This is termed as the key-translation step.

The basis of this protocol is a challenge for Q to P if P can encrypt the nonce n with the secret key that it shares with server S. The protocol correctness rests on S's trustworthiness – that S will properly decrypt using P's key and reencrypt using Q's key. The initialization and storage problems are greatly alleviated because each principal needs to keep only one key. The risk of compromise is mostly shifted to S, whose security can be guaranteed by various measures, such as encrypting stored keys using a master key and putting S in a physically secure room.

5.8 Password-based authentication

The use of passwords is a highly popular technique to achieve authentication because of low cost and convenience. This section is concerned with authentication techniques that are based on passwords. A problem with passwords is that people tend to pick a password that is convenient, i.e., short and easy to remember. Such passwords are vulnerable to a password-guessing attack, which works as follows: an adversary builds a database of possible passwords, called a dictionary. The adversary picks a password from the dictionary and checks if it works. This may amount to generating a response to a challenge or decrypting a message using the password or a function of the password. After every failed attempt, the adversary picks a different password from the dictionary and repeats the process. This non-interactive form of attack is known as the *off-line dictionary attack*.

Preventing off-line dictionary attacks

Thus, a major problem is that users tend to choose weak passwords, which are chosen from a sample space small enough to be enumerated by an adversary. Hence, protocols that are stronger than simple challenge–response protocols are needed to use these cryptographically weak passwords to securely authenticate entities. A password-based authentication protocol aims at preventing off-line dictionary attacks by producing a cryptographically strong shared secret key, called the session key, after a successful run of the protocol. This session key can be used by both entities to encrypt subsequest messages for a secret session.

In this section, we focus on protocols designed to prevent off-line dictionary attacks on password-based authentication. Next, we present two password based authentication protocols.

5.8.1 Encrypted key exchange (EKE) protocol

The first attempt to protect a password protocol against off-line dictionary attacks was made by Bellovin and Merritt who developed a password based encrypted key exchange (EKE) protocol using a combination of symmetric and asymmetric cryptography. Algorithm 16.15 describes the EKE protocol that works as follows: suppose users A and B are participating in a run of the protocol. (Recall that X,k denotes the encryption of X using a symmetric key *k* and Y,k−1 denotes the decryption of Y using a symmetric key *k*.)

In step 1, user A generates a public/private key pair (EA,DA) and also derives a secret key Kpwd from his/her password pwd. In step 2, A encrypts his/her public key EA with Kpwd and sends it to B. In steps 3 and 4, B decrypts the message and uses EA together with Kpwd to encrypt a session key KAB and sends it to A. In steps 5 and 6, A uses this session key to encrypt a unique challenge CA and sends the encrypted challenge to B. In step 7, B decrypts the message to obtain the challenge and generates a unique challenge CB. In step 8, B then encrypts {CA, CB} with the session key KAB and sends it to A. In step 9, A decrypts this message to obtain CA and CB and compares the former with the challenge it had sent to B. If they match, the correctness of B's response is verified (i.e., B is authenticated). In step 10, A encrypts B's challenge CB with the session key KAB and sends it to B. When B receives this message, it decrypts the message to obtain CB and uses it verify the correctness of A's response and to authenticate A. Note that the protocol results in a session key (stronger than the shared password) which the users can later use to encrypt sensitive data. The EKE protocol suffers from the plain-text equivalence, which means that the user and the host have access to the same secret password or hash of the password.

5.9 Authentication protocol failures

Despite the apparent simplicity of the basic design principles, realistic authentication protocols are notoriously difficult to design. There are several reasons for this:

• First, most realistic cryptosystems satisfy algebraic additional identities. These extra properties may generate undesirable effects when combined with a protocol logic.

• Second, even assuming that the underlying cryptosystem is perfect, unexpected interactions among the protocol steps can lead to subtle logical flaws.

• Third, assumptions regarding the environment and the capabilities of an adversary are not explicitly specified, making it extremely difficult to determine when a protocol is applicable and what final states are achieved.

We illustrate the difficulty by showing an authentication protocol proposed, with a subtle weakness. Consider the authentication protocol shown in Algorithm (kp and kq are symmetric keys shared between P and A, and Q and A, respectively, where A is an authentication server and k is a session key).

The message $\{k(P\}_{kQ}$ in step 3 can only be decrypted by Q and hence can only be understood by Q. Step 4 reflects Q's knowledge of k, while step 5 assures Q of P's knowledge of k; hence the authentication handshake is based entirely on the knowledge of k.

The subtle weakness in the protocol arises from the fact that the message k(P)kQ sent in step 3 contains no information for Q to verify its freshness. This is the first message sent to Q about P's intention to establish a secure connection. An adversary who has compromised an old session key k' can impersonate P by replaying the recorded message k_p,k_Q in step 3 and subsequently executing steps 4 and 5 using k'.

To avoid protocol failures, formal methods may be employed in the design and verification of authentication protocols. A formal design method should embody the basic design principles. For example, informal reasoning such as "if you believe that only you and Bob know k, then you should believe any message you receive encrypted with k was originally sent by Bob" should be formalized by a verification method.

(1) $P \rightarrow A : \quad P, Q, n_p$

(2) $A \rightarrow P : \quad \{n_p, Q, k\ , \{k, P\}_{k_Q}\}_{k_p}$

(3) $P \rightarrow Q : \quad \{k, P\}_{k_Q}$

(4) $Q \rightarrow P : \quad \{n_Q\}_K$

(5) $P \rightarrow Q : \quad \{n_Q + 1\}_K$

Karpagam Academy of Higher Education
Department of Computer Applications
Subject Code/Name : Elective:17CAP505N/Distributed Computing

Objective Type Questions

	UNIT 5							
S.No	**Questions**	**OPTION 1**	**OPTION 2**	**OPTION 3**	**OPTION 4**	**OPTION 5**	**OPTION 6**	**Answer Key**
1	Distributed shared memory is abstraction provided tothe programmer of a _____.	distributed system	DC	detection	termiantion			**distributed system**
2	Programmers access the data across the network using only read and write primitives as they would in a _____.	multiprocessors	single processors	mainframes	uniprocessor system			**uniprocessor system**
3	A part of each computer's memory is earmarked for shared space and the remainder is _____.	public memory	key generation	private memory	public key			**private memory**
4	There is no _____presented by a single memory access bus.	Deadlocks	bottlenecks	termination	detectition'			**bottlenecks**
5	DSM effectively provides a_____.	large main memory	virtual memory	main memory	no memory			**large main memory**
6	The main point of allowing concurrent access to the same data object is to_____.	decrease throughput	increase throughput	increase time	decrease time			**increase throughput**
7	As DSM is implemented under the covers using asynchronous message passing the overheads incurred are atleast as high as those of a _____ implementation.	message passing	message looping	message acceptance	message avoidance			**message passing**
8	Determining what semantics to allow for concurrent access to _____.	unshared objects	shared objects	shared time	unshared time			**shared objects**
9	Determining the best way to implement the _____of concurrent access to shared data.	syntax	regulations	semantics	acceptance			**semantics**
10	_____ can range from tightly coupled multicomputers to wide area DS with heterogeneous hardware and software.	DS	DC	mainframes	DSM systems			**DSM systems**
11	_____ is the ability of the system to execute memory operations correctly.	memory abstraction	memory consistency model	Memory coherence	memory locking			**Memory coherence**
12	The _____ defines the set of allowlable memory access orderings.	memory abstraction	memory consistency model	Memory coherence	memory locking			**memory consistency model**
13	The strict consistency model also known as the _____	atomic consistency model.	memory consistency model	Memory coherence	memory locking			**atomic consistency model.**
14	An alternate way of specifying the consistency model in terms of the_____ and response to each read and write operation.	invocation	shared objects	shared time	unshared time			**invocation**
15	Implementing linearability is _____ because a global time scale needs to be simulated.	inexpensive	expensive	very low	low			**expensive**

16	_____ or strict atomic consistency is difficult to implement because the absence of a global time reference in a DS necessistates that the time reference has to be simulated.	unstricted	labelled	Linearability	unlabelled			**Linearability**
17	The first weaker model that of sequential consistency was proposed by Lamport and uses logical time reference instead of the _____	localtime	global time	reference time	global time reference.			**global time reference.**
18	An_____ is less restrictive than linearability it should be easier to implement.	locally consistency	global consistency	parallel consistency	sequential consistency			**sequential consistency**
19	At a proceesor the serial order of the events defines the_____.	global casual order	non order	local casual order	pre order			**local casual order**
20	A _____ casually precedes a read operation issued by another processor if the read returns a value written by the write.	read operation	write operation	read/write operation	no operation			**write operation**
21	The transitive closure of the read and write operation relations defines the _____.	global causal order	non order	local casual order	pre order			**global causal order**
22	Pipelined memory is otherwise called as _____.	locally consistency	processor consistency	parallel consistency	sequential consistency			**processor consistency**
23	_____ requires all causually realted Writes to be seen in the same order by all processes.	locally consistency	processor consistency	Casual consistency	sequential consistency			**Casual consistency**
24	In realtion to the cusality realtion between _____ only the local causality relation as defined by the local order write operations needs to be seen by other processors .	syntax	operations	semantics	acceptance			**operations**
25	The next weaker consistency model is that is _____.	slow memory	high memory	medium usage	low memory			**slow memory**
26	Slow memory can be implemented using a broadcast primitive that is weaker than even the _____	FIFO broadcast	LIFO model	FIFO model	no model			**FIFO broadcast**
27	The consistency models seen so far apply to all the instructions in the _____	undistributed program	distributed program.	local checkpoints	global checkpoints			**distributed program.**
28	Examples of consistency models based on the principle are erntry consistency, weak consistency and _____.	unreleased consistency	global consistency	release consistency	sequential consistency			**release consistency**
29	The synchronization _____ are inserted in the program based on the semantics of the types of the accesses.	syntax	operations	semantics	statements			**statements**
30	A _____ in the model has the sematics such as used to propagate all writes to other processors.	asynchronization	invariable	synchronization variable	detection			**synchronization variable**
31	A _____of the release consistency model is called the lazy release consistency model.	detection	relaxation	avoiding	mutually detected			**relaxation**
32	_____ requires the programmer to use acquire and release at the startand end at each CS, respectively.	Entry consistency	processor consistency	Casual consistency	sequential consistency			**Entry consistency**
33	OS have traditionally dealt with multi process synchronization using algorithms based on the prinicples high level constructs such as _____.	semaphores	semaphores and monitors	monitors	tasks			**semaphores and monitors**

34	Special automatically executed instructions supported by special purpose _____.	software	middleware	hardware	harddisk			**hardware**
35	Lamport proposed the classical bakery algorithmfor n_porcess mutual exclusion in _____	non shared memory	virtual memory	main memory	shared memory systems.			**shared memory systems.**
36	The algorithm can be shown to satisfy the requirements of the _____ problem.	local section	global section	critical section	infinite time			**critical section**
37	Time complexity in many environments the level of the contention may be low of _____.	O(MN)	O(mn)	O(m)	O(n)			**O(n)**
38	_____ is a property that gurantees that any process can complete any synchronization operation in a finitenumber of lower level steps irrespective of the execution speed of other processes.	free	await	Wait freedom	stop			**Wait freedom**
39	A _____ that does not overlap with a write operation returns that most recent value written to that register.	write	read operation	write read	readwrite			**read operation**
40	Regular register in addition to being a safe register a read that is concurrent with a write operation returns either the value before the write operation or the value written by the_____.	write operation	read operation	write read	readwrite			**write operation**
41	Atomic register in addition to being a regular register te register is _____-.	unlinearizable	linearizable	acceptable	unacceptable			**linearizable**
42	A study of register construction deals with designing the more complex registers using _____.	stack registers	memory registers	simple registers	local registers			
43	In a_____ we must not only restore the system to aconsistent state but also appropriately handle messages that are left in an abnormal state due to the failure and recovery.	failure free	failure occurence	fail_stop manner	failure recovery			**failure recovery**
44	_____ failures further complicate the recovery process.	consistent	recovery approach	Overlapping	availability			**Overlapping**
45	In the checkpoint based _____ the state of each process and the communication channel is checkpointed frequently so that upon a failure the system can be restored toa globally consistent set of checkpoints.	consistent	recovery approach	persistence	availability			**recovery approach**
46	In_____ each process has autonomy in deciding when to take checkpoints.	uncoordinated checkpointing	checkpoints independently	checkpoints dependently	blocking			**uncoordinated checkpointing**
47	In coordinated checkpointing processes orchestrate their checkpointing activities so that all local checkpoints from a _____.	inconsistent checkpoints	consistent global state	local checkpoints	local states			**consistent global state**
48	Coordinated checkpointing simplfies recovery and is not sueptible to the domino effect, since every process always restarts from its most _____.	executed checkpoint	independent checkpoints	recent checkpoint	local checkpoints			**recent checkpoint**
49	A straightforward appraoch to coordinated checkpointing is to block communications while the checkpointing protocol _____.	blocks	processes	terminates	executes			**executes**

50	A fundamental problem in cocordinating checkpointing is toprevent a process from receiving application messages that could make the check point _____.	consistent	persistence	availability	inconsistent			**inconsistent**
51	Coordinated checkpointing requires all processes to participate in every _____.	transmission	Detection	checkpoint	corrections			**checkpoint**
52	A min process _____ is one that forcesonly a minimum number of processes to take new checkpoint and at the same time it does not force any process to suspend its computation.	checkpoints independently	nonblocking checkpointing algorithm	checkpoints dependently	blocking			**nonblocking checkpointing algorithm**
53	Communication induced checkpointing is another way to avoid the domino effectwhile allowing processes to take some of their _____.	checkpoints independently	checkpoints dependently	nonblocking	blocking			**checkpoints independently**
54	A log based rollback recovery makes use of deterministic and non deterministic events in a _____.	termination	computation	processing	scheduling			**computation**
55	A successful identity authentication results in a belief held by the authenticating prinicpal that the authenticated possessesthe claimed _____.	tasks	identity	looping	processing			**identity**
56	In a symmetric cryptosystem knowing the shared key lets a principal encrypt andecrypt_____	local messages	global messages	arbitrary messages.	transferring messages			**arbitrary messages.**
57	The wide mouth frog protocol uses similar approach where a principal A _____ to principal B using the server S.	termiates itself	activates itself	process itself	authenticates itself			**authenticates itself**
58	The Otway Rees protocol is a server based protocol that provides _____ only in four messages without requiring timestamps.	non secure	secure	authenticated key transport	private key			**authenticated key transport**
59	The goal of Project Athena ws to create an _____based on high performance workstations,high speed networking and serversof various types.	service	educational computing environment	transporting	media			**educational computing environment**
60	The Needham Schroeder _____uses a trusted key server that issues certificates containing the public key of a user.	public key protocol	private key protocol	public authentication protocol	private authetication			**public key protocol**