CHAPTER

Concurrency and cloud computing

10

Concurrency is at the heart of cloud computing, large workloads generated by many cloud applications run concurrently on multiple instances, taking advantage of ample resources provided by a cloud infrastructure. The practical motivations for concurrent execution are to: (i) overcome physical limitations of any single system by distributing the workload among several servers and (ii) significantly reduce the completion time of a computation.

Leslie Lamport began his 2013 Turing Lecture dedicated to Edsger Dijkstra [294] with the observation that concurrency has been known by several names: "I don't know if concurrency is a science, but it is a field of computer science. What I call concurrency has gone by many names, including parallel computing, concurrent programming, and multiprogramming. I regard distributed computing to be part of the more general topic of concurrency."

Is there a distinction between *concurrency* and *parallel processing?* According to some, concurrency describes the necessity that multiple computations are executed at the same time, while parallel processing implies a solution, there are multiple physical systems capable of carrying out the computations required by these activities at the same time, i.e., concurrently. Concurrency emphasizes cooperation and interference among activities, while parallel processing aims to shorten the completion time of the set of activities, and it is hindered by cooperation and activity interference.

Execution of multiple activities in parallel can proceed either quasi-independently, or tightly coordinated with an explicit communication pattern. In either case, some form of communication is necessary for coordination of concurrent activities. Coordination complicates the description of a complex activity because it has to characterize the work done by individual entities working in concert, as well as the interactions among them.

Communication affects overall efficiency of concurrent activities and could significantly increase the completion time of an application with a large number of concurrent tasks communicating frequently. Furthermore, communication requires prior agreement on the communication discipline described by a communication protocol. Measures to ensure that anomalies, e.g., deadlocks, do not affect the overall orchestration required by the application are also necessary.

The chapter starts with a brief discussion of concurrency's enduring challenges in Section 10.1 and continues with an overview of concurrent execution of communicating processes in Section 10.2, followed by a discussion of computational models in Section 10.3. BSP, a bridging hardware-software model designed to avoid logarithmic losses of efficiency in parallel processing, and its version for a multicore computational model are covered in Sections 10.4 and 10.5. Petri nets, discussed in Section 10.6, are intuitive models able to describe concurrency and conflict.

The concept of process state, critical for understanding concurrency, is covered in Section 10.7. Many functions of a computer cloud require information about process state. For example, controllers for cloud resource management discussed in Chapter 9 require accurate state information. Process coordination is analyzed in Section 10.8, while Section 10.9 presents logical clocks and message delivery rules in an attempt to bridge the gap between the abstractions used to analyze concurrency and the physical systems.

The concept of consistent cuts and distributed snapshots are at the heart of *checkpoint–restart* procedures for long-lasting computations. Checkpoints are taken periodically in anticipation of the need to restart a software process when one or more systems fail; when a failure occurs, the computation is restarted from the last checkpoint, rather than from the beginning. These concepts are discussed in Sections 10.10 and 10.11. Atomic actions, consensus protocols, and load balancing are covered in Sections 10.12, 10.13, and 10.14, respectively. Section 10.15 presents multithreading and concurrency in Java, FlumeJava, and Apache Crunch.

This chapter reviews theoretical foundations of important algorithms at the heart of system and application software. The concepts introduced in the next sections help us better understand cloud resource management policies and the mechanisms implementing these policies. For a deeper understanding of the many subtle concepts related to concurrency, the reader should consult the classical references discussed at the end of the chapter.

10.1 Enduring challenges

Concurrency is a very broad subject, and in this chapter we restrict the discussion to topics closely related to cloud computing. We start with a gentle introduction to some of the enduring challenges posed by concurrency and coordination. Coordination starts with resource allocation to the units carrying out individual tasks and workload distribution among these units. The initial phase is followed by communication during execution of the tasks, and finally, by the assembly of individual results. Coordination is ubiquitous in our daily life, and the lack of coordination has implications for the results. For example, Fig. 10.1 shows that lack of coordination and disregard of the rules regarding the management of shared resources lead to traffic deadlock, an unfortunate phenomenon we often experience.

Synchronization[1] is an important aspect of concurrency illustrated by the dining philosophers problem, see Fig. 10.2. Five philosophers sitting at a table alternately think and eat. To eat a philosopher needs the two chopsticks placed left and right of her plate. After finishing eating she must place the chopsticks back on the table to give a chance to her left and right neighbors to eat. The problem is nontrivial, the naive solution when each philosopher picks up the chopstick to the left and waits for the one to the right to become available, or vice versa, fails because it allows the system to reach a deadlock when no progress is possible. Deadlock would lead to philosopher starvation, a state to be avoided.

This problem captures critical aspects of concurrency, such as mutual exclusion and resource starvation discussed in this chapter. Edsger Dijkstra proposed the following solution formulated in terms of resources in general:

[1] Synchronization refers to one of two distinct concepts: process synchronization and data synchronization. Process synchronization, discussed in this section, means that multiple processes handshake at a certain point to reach an agreement or to commit to a sequence of actions. Data synchronization means keeping multiple copies of a dataset in coherence with one another, or maintaining data integrity.

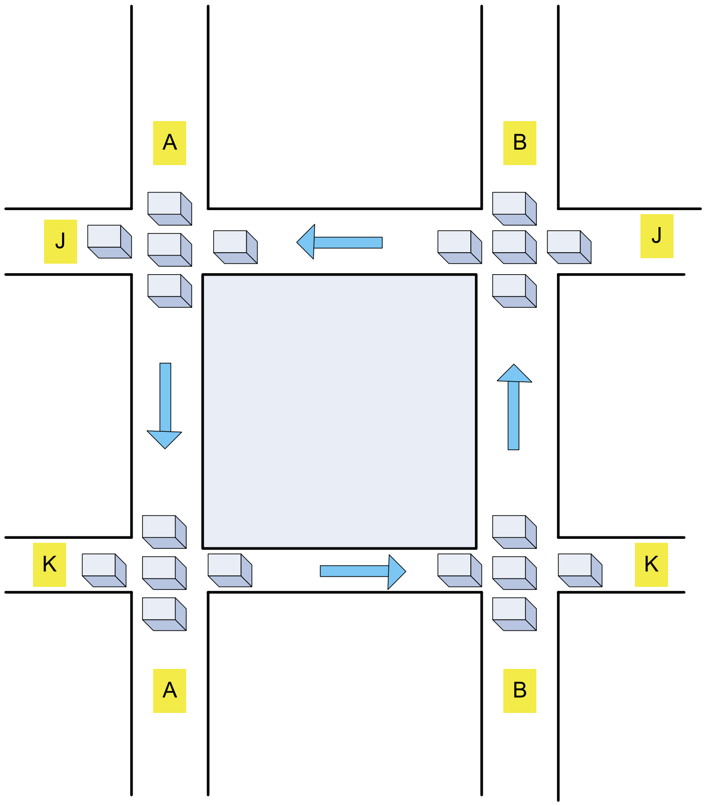

FIGURE 10.1

The consequences of the lack of coordination. The absence of traffic lights or signs in intersections causes a traffic jam. Blocking the intersections, A-J, A-K, B-J, and B-K, shared resources for North–South and East–West traffic flows, creates deadlocks.

i. Assign a partial order to the resources.
ii. Impose the rule that resources must be requested in order.
iii. Impose the rule that no two resources unrelated by order will ever be used by a single unit of work at the same time.

In the dining philosopher problem, the resources are the chopsticks, numbered 1 through 5, and each unit of work, i.e., philosopher, will always pick up the lower-numbered chopstick first and then the higher-numbered one next to her. The order in which each philosopher puts down the chopsticks does not matter.

This solution avoids starvation. If four of the five philosophers pick up their lower-numbered chopstick at the same time, only the highest-numbered chopstick will be left on the table. Therefore, the fifth philosopher will not be able to pick up a chopstick. Only one philosopher will have access to that

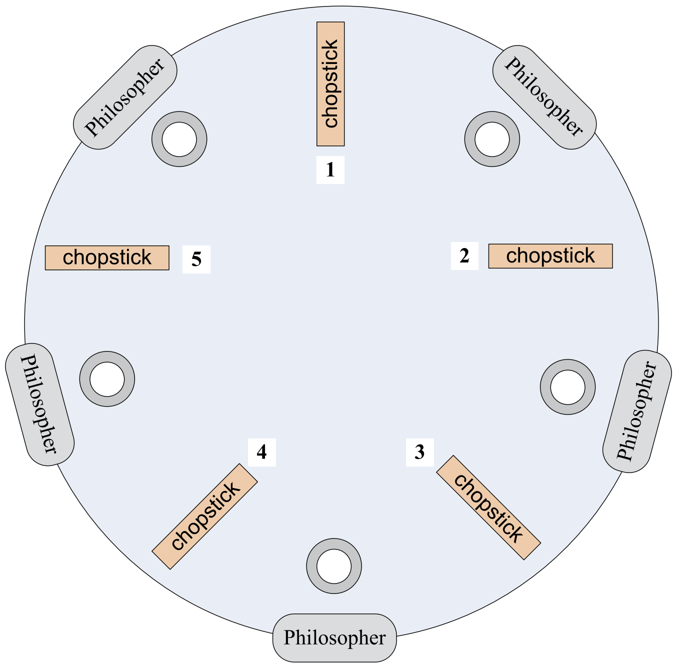

FIGURE 10.2

Dining philosophers problem. To avoid deadlock Dijkstra's solution requires numbering of chopsticks and two additional rules.

highest-numbered chopstick, so she will be able to eat using two chopsticks. Reference [309] presents a solution of the dining philosopher problem based on a Petri Net model.

The division of work comes naturally when some activities of a complex task require special competence and can only be assigned to uniquely qualified entities. In other cases, all entities have the same competence and the work should be assigned based on the individual ability to carry out a task more efficiently. Balancing the workload could be difficult in all cases because some activities may be more intense than others.

Though concurrency reduces the time to completion, it can negatively affect the efficiency of individual entities involved. Sometimes, a complex task consists of multiple stages and transitions from one stage to the next can only occur when all concurrent activities in one stage have finished their assigned work. In this case, the entities finishing early have to wait for the others to complete, an effect called *barrier synchronization*.

This discussion shines some light on the numerous challenges inherent to concurrency. Many computational problems are rather complex, and concurrency has the potential to greatly affect our ability to compute efficiently. This motivates our interest in concurrency and its applications to cloud computing.

Parallel and distributed computing exploit concurrency and have been investigated since mid-1960s. Parallel processing refers to concurrent execution on a system with a large number of processors, while distributed computing means concurrent execution on multiple systems, often at different locations. Communication latency is considerably lower in the first case, while distributed computing could only be efficient for coarse-grained parallel applications when concurrent activities seldom communicate with one another. Metrics such as execution time, speedup, and processor utilization discussed in Chapter 3 characterize how efficiently a parallel or distributed system can process a particular application.

Topics discussed in this chapter, such as computational models, checkpointing, atomic actions, and consensus algorithms, are relevant to both parallel and distributed computing. Multithreading is more relevant to parallel processing, while load balancing is particularly important to distributed systems.

This distinction is blurred for computer clouds because their infrastructure consists of millions of servers at one data center, possibly linked by high-speed networks with computers at another data center of the same cloud service provider. Nevertheless, communication latency is a matter of concern in cloud computing as we can see in Chapters 4, 9, and 11.

10.2 Communication and concurrency

Concurrency implies cooperative execution of multiple processes/threads in parallel. Concurrency can be exploited for minimizing the completion time of a task, while maximizing efficiency of the computing substrate. Computing substrate is a generic term for the physical systems used for application processing. To analyze the benefits of concurrency, we consider the decomposition of a computation into virtual tasks and relate them to the physical processing elements of the computing substrate.

The larger the cardinality of the set of virtual tasks, the higher the degree of concurrency, thus the higher the potential speedup. A parallel algorithm can then be implemented by a parallel program able to run on a system with multiple execution units. A *process* is a program in execution and requires an *address space* hosting the code and the data of an application. A *thread* is a lightweight execution unit running in the address space of a process.

The *speedup* of concurrent execution of an application quantifies the effect of concurrent execution, and it is defined as the ratio of the completion time of sequential execution of the task versus the concurrent execution completion time. For example, Fig. 10.3 illustrates concurrent execution of an application in which the workload is partitioned and assigned to five different processors or cores running concurrently. The application is the conversion of 5×10^6 images from one format to another. This is an *embarrassingly parallel* application because the five threads running on five cores, each processing 10^6 images, do not communicate with one another. The speedup S for the example in Fig. 10.3 is

$$S = \frac{t_8 - t_0}{t_3 - t_0} \approx 5. \tag{10.1}$$

There are two sides of concurrency, algorithmic or logical concurrency, discussed in this chapter, and physical concurrency, discovered and exploited by the software and the hardware of the computing substrate. For example, a compiler can unroll loops and optimize sequential program execution, and a processor core may execute multiple program instructions concurrently, as discussed in Chapter 3.

Concurrency is intrinsically tied to communication; concurrent entities must communicate with one another to accomplish the common task. The corollary of this statement is that communication and

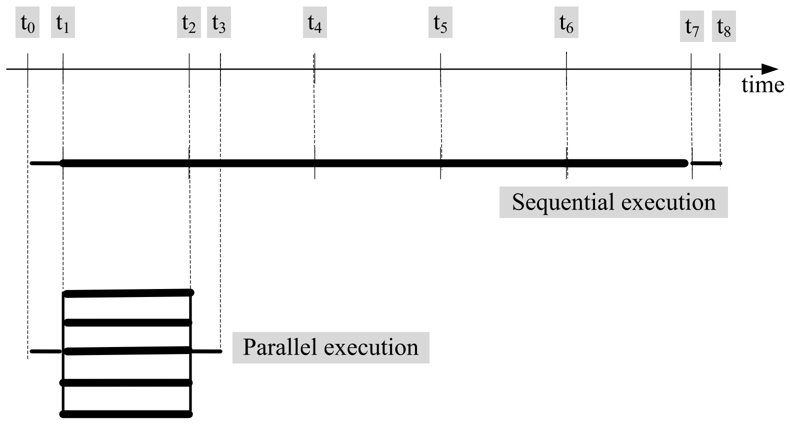

FIGURE 10.3

Sequential versus parallel execution of an application. The sequential execution starts at time t_0, goes through a brief initialization phase until time t_1 and then starts the actual image processing. When all images have been processes, it enters a brief termination phase at time t_7, and finishes at time t_8. The concurrent execution has its own brief initialization and termination phases; the actual image processing starts at time t_1 and ends at time t_2. The results are available at time $t_3 \ll t_8$. The speedup is close to the maximum speedup.

computing are deeply intertwined. Explicit communication is built into the blueprint of concurrent activities; we shall call it *algorithmic communication* for lack of a better term. A multithreaded application consists of one or more *thread groups*, sets of threads that work in concert and have to coordinate with each other.

Sometimes, a multithreaded computation consists of multiple stages when concurrently running threads of a thread group cannot continue to the next stage until all of them have finished the current one. This leads to inefficiencies, as shown in Fig. 10.4, that illustrates the effects of *barrier synchronization*. Such thread groups (or process groups) have to be scheduled together to minimize the blocking time, a process called *co-scheduling* [34].

Communication is a more intricate process than the execution of a computational step on a machine that rigorously follows a sequence of instructions from its own repertoire. Besides the sender and the receiver(s), communication involves a third party, a communication channel that may, or may not be reliable. Therefore, the two or more communicating entities have to agree on a communication protocol consisting of multiple messages. *Communication complexity* reflects the amount of communication that the participants of a communication system need to exchange to be aböle to perform certain tasks [525].

Fig. 10.5 illustrates the case when short bursts of computations alternate with relatively long periods when a thread is blocked, waiting for messages from other threads, the so-called *fine-grained* parallelism. The opposite is *coarse-grained* parallelism when little or no communication among concurrent threads takes place, as can be seen in Fig. 10.3.

Communication speed is considerably slower than computation speed. A processor could execute billions of instructions during the time it is blocked waiting to receive a message. It is thus expected that an application exhibiting fine-grained parallelism, i.e., communicating frequently, experiences long

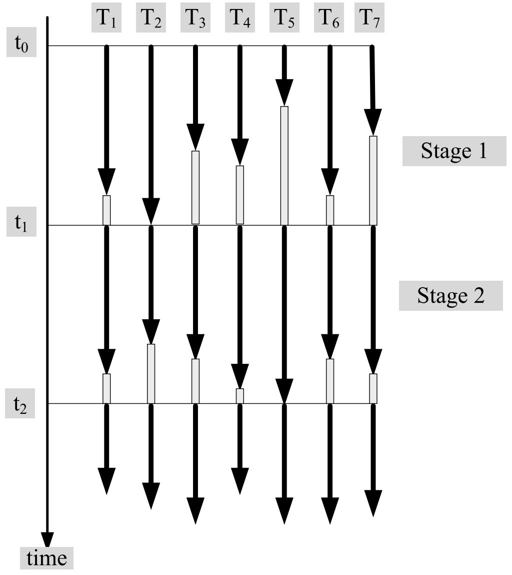

FIGURE 10.4

Barrier synchronization. The computation involving a group of seven threads running concurrently consists of multiple stages; all threads in the thread group must finish execution of one stage before proceeding to the next. All threads start execution of Stage 1 at time t_0. Threads T_1, T_3, T_4, T_5, T_6, and T_7 finish early and have to wait for thread T_2 before proceeding to Stage 2 at time t_1. Similarly, threads T_1, T_2, T_3, T_4, T_6, and T_7 have to wait for thread T_5, before proceeding to the next stage at time t_2. Black arrows indicate running threads, while white bars represent blocked tasks, waiting to proceed to the next stage.

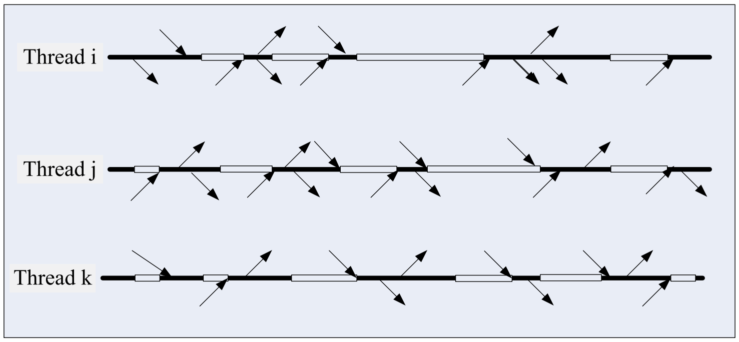

FIGURE 10.5

Fine-grained parallelism. Short bursts of computations of three concurrent threads are interspaces with blocked periods when a thread is waiting for messages from other threads. Solid black bars represent running threads, while white bars represent blocked threads waiting for messages. Arrows represent messages sent or received by a thread.

blocking periods. Intensive communication can slow down considerably a group of concurrent threads of an application, especially in case of multistage computations with frequent barrier synchronizations.

The word *message* should be understood in an information theoretical sense, rather than the more narrow meaning used in computer networks context. Embarrassingly parallel activities can proceed without message exchanges and enjoy linear or even superlinear speedup, while in all other cases, the completion time of communicating activities is increased because the communication bandwidth is significantly lower than the processor bandwidth.

Nonalgorithmic communication is required by the organization of a computing substrate consisting of various components that need to act in concert to carry out the required task. For example, in a system consisting of multiple processors and memories, a thread running on one processor may require data stored in the memory of another one. The nonalgorithmic communication, unavoidable in a distributed systems, occurs as a side effect of thread scheduling and data distribution strategies and can dramatically reduce the benefits of concurrency.

Spatial and temporal locality affect the efficiency of a parallel computation. A sequence of instructions or data references enjoy *spatial locality* if the items referenced in a narrow window of time are close in space, e.g., are stored in nearby memory addresses, or nearby sectors on a disk. A sequence of items enjoy *temporal locality* if accesses to the same item are clustered in time.

Nonalgorithmic communication complexity may decrease due to locality; it is more likely that a virtual task may find the data it needs in the memory of the physical processor where it runs. Also, the efficiency of the computing substrate may increase because at any given time the scheduler may find sufficient read-to-run virtual tasks to keep the physical processing elements busy. The scale of a system amplifies the negative effects of both algorithmic and nonalgorithmic communication. The interaction between the virtual and physical aspects of concurrency gives us a glimpse at the challenges faced by a computational model of concurrent activities.

Today's computing systems implement two computational models, one based on *control flow* and the other one on *data flow*. The ubiquitous von Neumann model for sequential execution implements the former; at each step, the algorithm specifies the step to be executed next. In this model, concurrency can only be exploited through the development of a parallel algorithm reflecting the decomposition of a computational task into processes/threads that can run concurrently, while exchanging messages to coordinate their progress. In the case of control flow, the *if then else* construct is shown by the top graph and the *while* loop on the bottom graph in Fig. 10.6. Instructions I_2 and I_3 of the *if then else* construct can run concurrently only after instruction I_1 finishes. Instruction I_4 can only be executed when I_2 and I_3 finish execution.

The *data flow* execution model is based on an implicit communication model, where a thread starts execution as soon as its input data become available. The advantages of this execution model are self-evident; it effortlessly extracts maximum amount of parallelism from any computation.

The data flow model example in Fig. 10.6 shows a maze of computations $C_1, \ldots, C_{13}$ with complex dependencies. The data flow model allows all computations to run as soon as their input data become available. For example, C_1, $C_{3,}$ and C_4 start execution concurrently with input $data1$, $data3$, and $data4$, respectively, while C_2 and C_6 wait for completion of C_3 and C_1, respectively. Finally, C_{13} can only start when $data18$, $data12$, $data8$, $data13$, $data14$, $data16$, and $data17$ are produced by C_{10}, C_8, C_6, C_7, C_9, C_{11}, and C_{12} respectively.

The time required by computations C_i $1 \leq i \leq 13$ to finish execution and deliver data to the ones waiting depends upon the size of the input data which is not known a priori. To capture the dynamics of

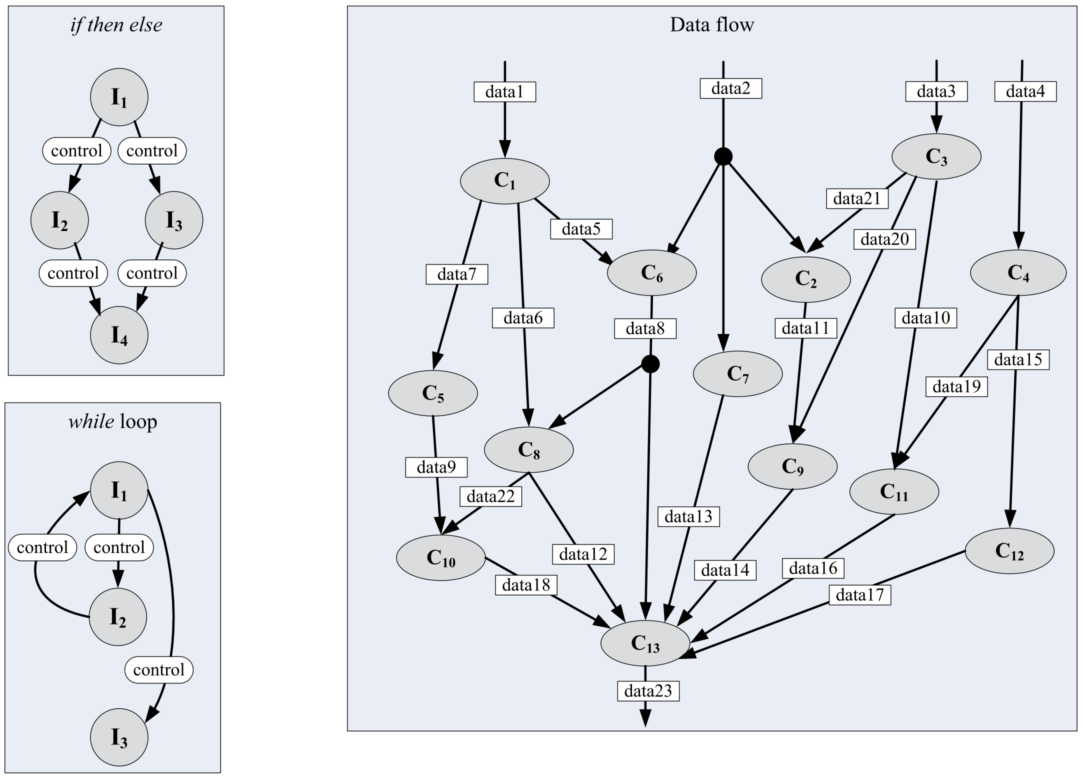

FIGURE 10.6

Control flow versus data flow models. Left graphs show the control flow from one instruction I_i, $1 \le i \le 4$ to the others for *if then else* and the *while* loop constructs. Either I_2 or I_3 will ever run in the *if then else* construct. The right graph illustrates the data flow model; now the arrival of the input data triggers the execution of each one of the computations C_i, $1 \le i \le 13$. For example, C_9 execution is triggered by the arrival of *data11* produced by C_2 and *data20* produced by C_3.

a computational task, the control flow model would require individual computations to send and receive messages, in addition to sending the data.

There are only a few data flow computer systems in today's landscape, but it is not beyond the realm of possibilities to see some added to the cloud computing infrastructure in the next future. Moreover, some of the frameworks for data processing discussed in Chapters 4 and 11 attempt to mimic the data flow execution model to optimize their performance.

Petri Nets models discussed in Section 10.6 are powerful enough to express either control flow or data flow. In these bipartite graphs one type of vertices, *places*, model system state, while the other type of vertices, *transactions*, model actions. Tokens flowing out of places trigger transactions and end up in other places, indicating a change of system state. Tokens may represent either control or data.

For several decades, concurrency was of interest mostly for systems programming and for high-performance computing in science and engineering. The majority of other application developers were content with sequential processing and expected increased computing power due to faster clock rates. Concurrency is now mainstream due to the disruptive effects of the multicore processor technology, the limitation imposed on the processor clock speed, combined with the insatiable appetite for computing power encapsulated in tiny and energy-frugal computing devices. Writing and debugging concurrent software is considerably more challenging than developing sequential code; it requires a different frame of mind and effective development tools.

The next three sections discuss computational models, abstractions needed to gain insight into fundamental aspects of computing and concurrency.

10.3 Computational models; communicating sequential processes

Several computational models are used in the analysis of algorithms. Each one of the two broad classes of models, *abstract machines* and *decision trees,* specify the set of primitive operations allowed. Turing machines, circuit models, lambda calculus, finite-state machines, cellular automaton, stack machines, accumulator machines, and random access machines are abstract machines used in proofs of computability and for establishing upper bounds on computational complexity of algorithms [439]. Decision tree models are used in proofs of lower bounds on computational complexity.

Computational models. Turing machines are *uniform models of computation*; the same computational device is used for all possible input lengths. Boolean circuits are *non-uniform models of computation*, inputs of different lengths are processed by different circuits. A computational problem $\mathcal{P}$ is associated with the family of Boolean circuits $\mathcal{C} = \{C_1, C_2, \dots, C_n, \dots\}$, where each C_n is a Boolean circuit handling inputs of n bits. A family of Boolean circuits $\mathcal{C}_n,\ n \in \mathbb{N}$, is *polynomial-time uniform* if there exists a deterministic Turing machine TM, such that TM runs in polynomial time and $\forall n \in \mathbb{N}$ it outputs a description of $\mathcal{C}_n$ on input 1^n.

A finite-state machine (FSM) consists of a logical circuit $\mathcal{L}$ and a memory $\mathcal{M}$. An execution step with the external input $L^{in} \in \Sigma$ takes the current state $S \in \mathcal{S}$ and uses the logic circuit $\mathcal{L}$ for producing a successor state $S^{new} \in \mathcal{S}$ and an output letter from the same alphabet, Σ, $L^{out} \in \Sigma$. FSMs are used for the implementation of sophisticated processing scheme such as TCP (Transmission Control Protocol) covered in Chapter 6 and used by the Internet protocol stack and the Zookeeper coordination model discussed in Section 11.4.

The operation of a serial computer using two synchronous interconnected FSMs, a central processing unit (CPU) with a small number of registers and a random-access memory, is modeled by a *random-access machine* (RAM). CPU operates on data stored in its registers. *Parallel random-access machine* (PRAM) is an abstract programming model consisting of a bounded set of RAM processors and a common memory of a potentially unlimited size. Each RAM has its own program and program counter and a unique Id. During a PRAM execution step, the RAMs execute synchronously three operations: read from the common memory, perform a local computation, and write to the common memory.

The von Neumann model, based on the von Neumann machine architecture, has been an enduring model of sequential computations. It has continued to allow “a diverse and chaotic software to run efficiently on a diverse and chaotic world of hardware” [485]. The model has endured the rapid pace of technological changes since 1947 when ENIAC performed the first Monte Carlo simulations for the Manhattan Project [221]. A brilliant feature of the von Neumann model is its clairvoyance, the ability to remain consistent in the face of later concepts such as memory hierarchies, certainly not available in the mid-1940s. This model has been the *zeitgeist*[2] of computing for more than half a century.

Communicating Sequential Processes (CSP). CSP is a formal language for describing patterns of interaction in concurrent systems proposed by Tony Hoare in 1978 [241]. CSP treats input and output as fundamental programming primitives and includes a simple form of parallel composition based on synchronized communication. CSP programs are a parallel composition of a fixed number of sequential named processes communicating with each other only through synchronous message-passing. Each message is specified by the name of the intended sending or receiving process. The concept of a *process* is a natural abstraction of the way parallel activities behave.

Brookes, Hoare, and Roscoe refined the theory of CSP into a process algebraic form [74]. CSP process algebra includes two classes of primitives, *initial events*, indivisible and instantaneous primitives representing interactions, and *primitive processes,* representing actions/behaviors. Several algebraic operators combine events and processes:

prefix—creates a new process by combining an event and a process, e.g., $a \rightarrow P$ describes a process like P willing to use primitive event a to communicate with the environment.
hiding—makes event a unobservable to primitive process P, $(a \rightarrow P) \setminus \{a\}$.
interleaving—a new process that behaves like primitive process P or Q simultaneously; the operator $|||$ represents independent concurrent activities.
deterministic choice—a new process is defined as a choice between two processes P and Q chosen by the environment which communicates its choice via two primitive events a and b, $(a \rightarrow P)\Box(b \rightarrow Q)$.
nondeterministic choice—a new process is defined as a choice between two processes P and Q, but the environment has no control over the choice, $(a \rightarrow P) \sqcap (b \rightarrow Q)$. The new process can behave as $(a \rightarrow P)$ or as $(b \rightarrow Q)$.

A fair number of tools to analyze systems described by CSP exist. For example, Failures/Divergence Refinement (FDR) is a family of model checker that converts two CSP process expressions into Labelled Transition Systems and then determines whether one of the processes is a refinement of the other within some specified semantic model; see http://www.fsel.com/.

CSP was used in 1998 to analyze the software, consisting of some 22 000 lines of code, for a fault-tolerant computer used to control the assembly, reboost[3] operations and flight control, and data management for experiments carried out by the International Space Station. The analysis confirmed that the design was deadlock-free [82].

[2] The German word "Zeitgeist" literally translated as "time spirit" is used to identify the dominant set of ideas and concepts of the society in a particular field and at a particular time.
[3] Reboost is the process of boosting the altitude of an artificial satellite, to increase the time until its orbit will decay and it reenters the atmosphere.

10.4 The bulk synchronous parallel model

Leslie Valiant developed in the early 1990s a bridging hardware-software model designed to avoid logarithmic losses of efficiency in parallel processing [485]. Valiant argues that *parallel and distributed computing should be based on a model emphasizing portability and the algorithms should be parameter-aware and designed to run efficiently on a broad range of systems with a wide variation of parameters.* The algorithms have to be written in a language that can be compiled effectively on a computer implementing the BSP model.

BSP is an unambiguous computational model, including parameters quantifying the physical constraints of the computing substrate. It also includes a nonlocal primitive, the barrier synchronization. The BSP model aims to free the programmer from managing memory, communication, and low-level synchronization, provided that programs are written with sufficient *parallel slackness*. Parallel slackness means hiding communication latency by providing each processor with a large pool of ready-to-run tasks, while other tasks are waiting for either a message or the completion of another operation.

BSP programs are written for v virtual parallel processors running on $p \leq v$ physical processors. This choice permits high-level language compilers to create executables sharing a virtual address space and to schedule and pipeline computation and communication efficiently. The BSP model includes:

i. Processing elements and memory components.
ii. A router involved in message exchanges between pairs of components; the throughput of the router is $\bar{g}$ and s is the startup cost.
iii. Synchronization mechanisms acting every L units of time.

The computation involves *supersteps* and *tasks*. A superstep is an execution unit allocated L units of time and consisting of multiple tasks. Each task is a combination of local computations and message exchanges. The computation proceeds as follows:

1. At the beginning of a superstep, each component is assigned a task.
2. At the end of the time allotted to the superstep, after L units of time since its start, a global check determines if the superset has been completed:
   2.1. If so, the next superstep is initiated;
   2.2. Else, another period of L units of time is allocated allowing the superset to continue its execution.

Local operations do not automatically slow down other processors. A processor can proceed without waiting for the completion of processes in the router or in other components when the synchronization is switched off. The model does not make any assumption about communication latency. The value of the periodicity parameter L may be controlled, and its lower bound is determined by the hardware, the faster the hardware the lower L, while its upper bound is controlled by the granularity of the parallelism, hence by the software.

The router of the BSP computing engine supports arbitrary *h-relations*, which are supersteps of duration $\bar{g} \times h + s$ when each component sends and receives at most h messages. It is assumed that h is large, and $\bar{g} \times h$ is comparable to s. When $g = 2\bar{g}$ and $\bar{g} \times h \geq s$, an h-relation will require at most $g \times h$ units of time.

Hash functions are used by the model to distribute memory accesses to memory units of all BSP components. The mapping of logical or symbolic addresses to physical ones must be efficient and distribute the references as uniformly as possible, and this can be done by a pseudo-random mapping. When a superstep requires p random accesses to p components, then a component will get, with high probability, $\log p/\log\log p$ accesses. If a superstep requires $p\log p$ memory accesses, then each component will get not more than about $3\log p$ accessed, and these can be sustained by a router in the optimal bound $\mathcal{O}(\log p)$.

The simulation of a parallel program with $v \geq p\log p$ virtual processors on a BSP computer with p components is discussed next. Each BSP computer component consists of a processor and a memory. Each one of the p components of the BSP computer is assigned v/p virtual processors. The BSP will then simulate one step of a virtual processor in one superstep, and each one of the p memory modules will get v/p references if the memory references are evenly spread. The BSP will execute a superstep in an optimal time of $\mathcal{O}(v/p)$.

The multiplication of two $n \times n$ matrices A and B using a standard algorithm is a good example of simulation on a BSP machine with $p \leq n^2$ processors. Each processor is assigned an $n/\sqrt{p} \times n/\sqrt{p}$ submatrix and receives $n/\sqrt{p} \times n/\sqrt{p}$ rows of A and $n/\sqrt{p} \times n/\sqrt{p}$ columns of B. Thus, each processor will carry out $2n^3/p$ additions and multiplications and send $2n^2/\sqrt{p} \leq 2n^3/p$ messages. If each processor sends $2n^2/\sqrt{p}$ messages, then the running time is affected only by a constant factor.

Concurrency can be implemented efficiently by replicating data when h is small. In the matrix multiplication example, $2n^2/p$ elements of matrices A and B can be distributed to each one of the p processors, and each processor replicates each of its elements $\sqrt{p}$ times and sends them to the $\sqrt{p}$ processors that need the entries. The number of messages sent by each processor is $2n^2\sqrt{p}$. When $g = \mathcal{O}(n\sqrt{p})$ and $L = \mathcal{O}(n^3/p)$, we have the optimal running time of $\mathcal{O}(n^3/p)$. In addition to matrix multiplication, it is shown that several other important algorithms can be implemented efficiently on a BSP computer.

Valiant concludes that the BSP model helps programmability for computations with sufficient slack because the memory and communication management required to implement a virtual shared memory can be achieved with only a constant factor loss in processor utilization [485]. Moreover, the BSP model can be implemented efficiently for different communication technologies and interconnection network topologies, e.g., for a hypercube interconnection network.

10.5 A model for multicore computing

Extracting an optimal multicore processor performance is very challenging. Most of these challenges are intrinsically related to the complexity and diversity of the computational engines because the performance of an application on one system may not translate on high performance on another. Developing parallel algorithms is in itself nontrivial for many applications of interest and the application performance may not scale with the problem size for various computation substrates.

The Multi-BSP [486] is a hierarchical multicore computing model with an arbitrary number of levels. The computing substrate of the model includes multiple levels of cache, as well as the size of the memory. A *depth-d* model is a tree of depth d with caches and memory as the internal nodes of the tree and the processors at the leaves.

A depth-d model requires $4d$ parameters. Level i, $1 \leq i \leq d$, has four parameters (p_i, g_i, L_i, m_i) that quantify the number of subcomponents, the communication bandwidth, the synchronization cost, and the memory/cache size, respectively. The model captures the unavoidable communication cost due

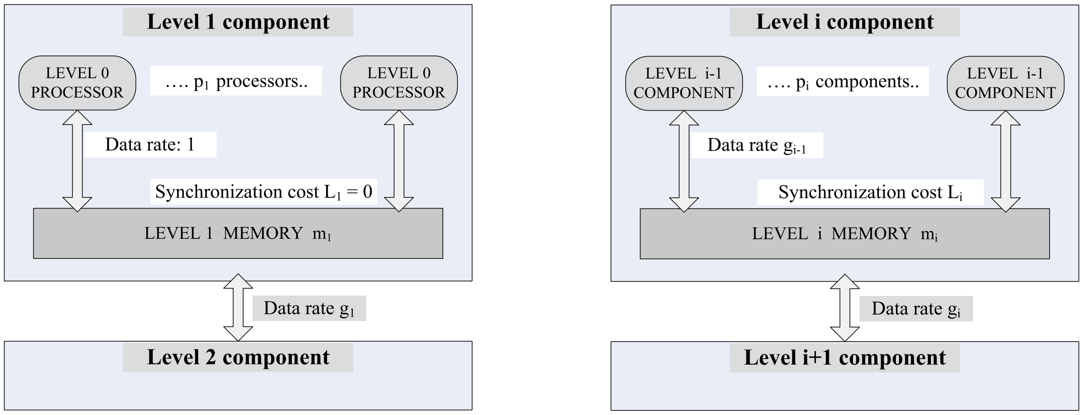

FIGURE 10.7

Multi-BSP model. A *depth-d* model is a tree of depth d with caches and memory as the internal nodes of the tree and the processors at the leaves. p_i—the number of $(i-1)$ level components inside an i-th level component; g_i—the communication bandwidth; L_i—the cost for barrier synchronization for a level-i superstep; m_i—the number of words of memory inside an i-th level component.

to latency L_i and bandwidth g_i as seen in Fig. 10.7. An in-depth description of level i parameters follows:

- p_i—the number of $(i-1)$ level components inside an i-th level component. The 1st level components consist of p_1 raw processors; a computation step on a word in level 1 memory represents one unit of time.
- g_i—the communication bandwidth, the ratio of the number of operations of a processor to the number of words transmitted between the memories of a component at level i and its parent component at level $(i+1)$, in a unit of time. A word represents the data the processor operates on. Level 1 memories can keep up with the processors; thus their data rate, g_0, is one.
- L_i—the cost for barrier synchronization for a level i superstep. The barrier synchronization is between the subcomponents of a component; there is no synchronization across separated branches in the component hierarchy.
- m_i—the number of words of memory inside an i-th level component that is not inside any $(i-1)$ level component.

The number of processors in a level-i component is $P_i = p_1 \cdot p_2 \cdot \ldots \cdot p_i$. The number of level-$i$ components in a level-j component is $Q_{i,j} = p_{i+1} \cdot p_{i+2} \cdot \ldots p_j$, and the number in the whole system is $Q_{i,d} = Q_i = p_{i+1} \cdot p_{i+2} \cdot \ldots \cdot p_d$. The total memory at level-i component is $M_i = m_i + p_i \cdot m_{i-1} + p_{i-1} \cdot p_i \cdot m_{i-2} + \ldots + p2 \cdot \ldots \cdot p_{i-1} \cdot p_i m_1$. The cost of communication from level 1 to outside level i is $G_i = g_i + g_{i-1} + \cdots + g_1$.

A level-i superstep is a construct within a level-i component that allows each of p_i-level $(i-1)$ components to execute independently until they reach a barrier. Only after reaching the barrier can all exchange information with the m_i memory of the level-i component at a communication cost of $(g_i -$

1). The communication cost is $m_{i-1}g_{i-1}$ where m_i is the number of words communicated between the memory at the i-th level and its level $(i-1)$ subcomponents. The model tolerates constant factors k_{comp}, k_{comm}, and k_{synch}, but for each depth d these constants are independent of the (p, g, L, m) parameters.

The question is whether a model with such a large number of parameters could be useful. It is shown that Multi-BSP algorithms for problems, such as matrix multiplication, FFT (Fast Fourier Transform), and comparison sorting, can be expressed as parameter-free [486]. A *parameter-free* version of an optimal Multi-BSP algorithm with respect to a given algorithm $\mathcal{A}$ means that it is optimal in:

1. Parallel computation steps to within multiplicative constant factors and in total computation steps to within additive lower order terms.
2. Parallel communication costs to within constant multiplicative factors among Multi-BSP algorithms.
3. Synchronization costs to within constant multiplicative factors among Multi-BSP algorithms.

The proofs of optimality of communication and synchronization in [486] are based on previous work on lower bounds on the number of communication steps of distributed algorithms.

10.6 Modeling concurrency with Petri nets

In 1962, Carl Adam Petri introduced a family of graphs for modeling concurrent systems, the Petri nets (PNs) [398]. PNs are used to model the dynamic, rather than the static system behavior, e.g., detect synchronization anomalies. PN models have been extended to study the performance of concurrent systems. More than 8 000 articles on the properties of different flavors of PNs and their applications have been published.

PNs are bipartite graphs populated with tokens flowing through the graph. A *bipartite graph* is one with two classes of vertices; arcs always connect a vertex in one class with one or more vertices in the other class. The two classes of PN vertices are *places* and *transitions,* thus the name Place-Transition (P/T) Nets often used for this class of bipartite graphs; arcs connect either one place with one or more transitions or a transition with one or more places.

Petri nets model the dynamic behavior of systems. The places of a PN contain tokens; firing of transitions removes tokens from the *input places* of the transition and adds them to its *output places*; see Fig. 10.8. PNs can model different activities in a distributed system. A *transition* may model the occurrence of an event, the execution of a computational task, the transmission of a packet, a logic statement, etc.

The *input places* of a transition model the preconditions of an event, the input data for the computational task, the presence of data in an input buffer, the preconditions of a logic statement. The *output places* of a transition model the postconditions associated with an event, the results of the computational task, the presence of data in an output buffer, or the conclusions of a logic statement.

The distribution of tokens in the places of a PN at a given time is called the *marking* of the net and reflects the state of the system being modeled. PNs are very powerful abstractions and can express both concurrency and choice as we can see in Fig. 10.9.

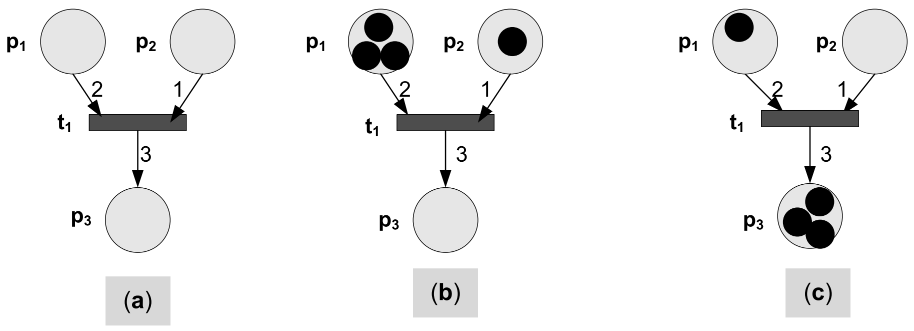

FIGURE 10.8

Petri nets firing rules. (a) An unmarked net with one transition t_1 with two input places, p_1 and p_2, and one output place, p_3. (b) The marked net, the net with places populated by tokens; the net before firing the enabled transition t_1. (c) The marked net after firing transition t_1, two tokens from place p_1 and one from place p_2 are removed and transported to place p_3.

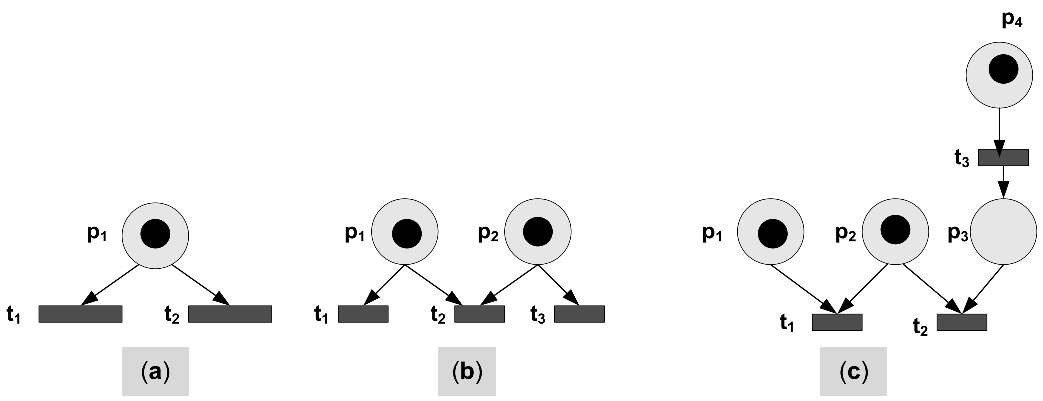

FIGURE 10.9

Petri nets modeling. (a) Choice; only one of transitions t_1 or t_2 may fire. (b) Symmetric confusion; transitions t_1 and t_3 are concurrent, and, at the same time, they are in conflict with t_2. If t_2 fires, then t_1 and/or t_3 is disabled. (c) Asymmetric confusion; transition t_1 is concurrent with t_3, and it is in conflict with t_2 if t_3 fires before t_1.

PNs can model concurrent activities. For example, the net in Fig. 10.9(a) models conflict or choice; only one of the transitions t_1 and t_2 may fire, but not both. Two transitions are said to be *concurrent* if they are causally independent. Concurrent transitions may fire before, after, or in parallel with each other; examples of concurrent transitions are t_1 and t_3 in Figs. 10.9(b) and (c).

When choice and concurrency are mixed, we end up with a situation called *confusion*. *Symmetric confusion* means that two or more transitions are concurrent and, at the same time, they are in conflict with another one. For example, transitions t_1 and t_3 in Fig. 10.9(b) are concurrent and, at the same time, they are in conflict with t_2. If t_2 fires, either one or both of them will be disabled. *Asymmetric confusion*

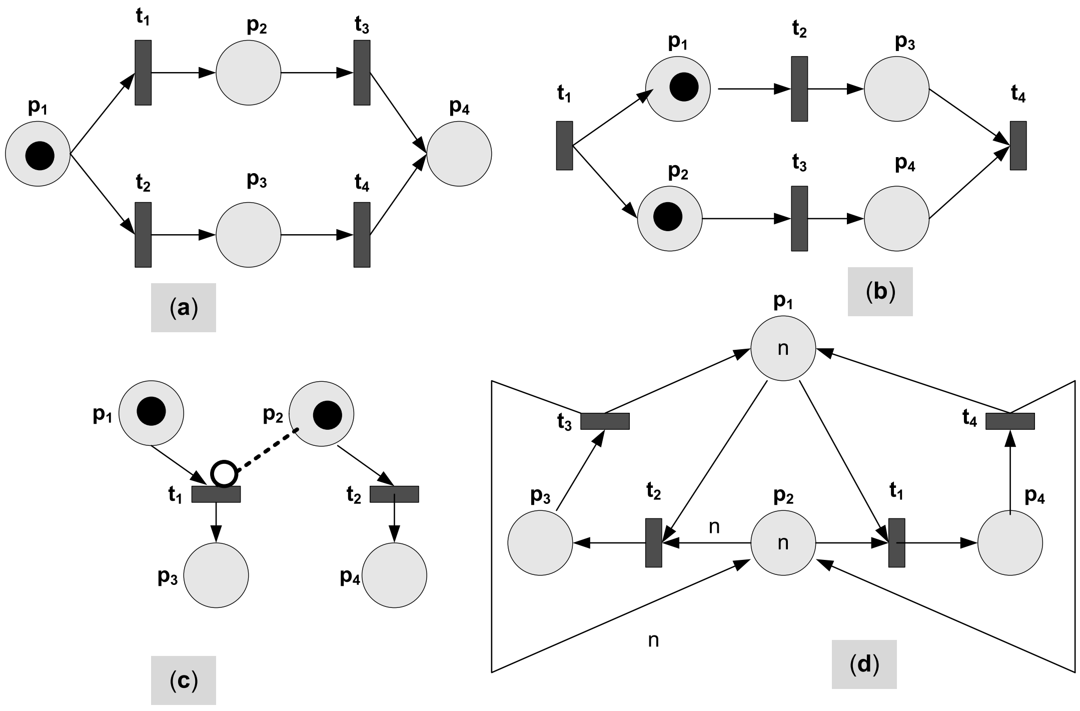

FIGURE 10.10

(a) A state machine; there is the choice of firing t_1, or t_2; only one transition fires at any given time; concurrency is not possible. (b) A *marked graph* can model concurrency but not choice; transitions t_2 and t_3 are concurrent; there is no causal relationship between them. (c) An extended net used to model priorities; the arc from p_2 to t_1 is an inhibitor arc. The process modeled by transition t_1 is activated only after the process modeled by transition t_2 is activated. (d) Modeling exclusion; transitions t_1 and t_2 model writing and, respectively, reading with n processes to a shared memory. At any given time only one process may write, but any subset of the n processes may read at the same time, provided that no process writes.

occurs when a transition t_1 is concurrent with another transition t_3 and will be in conflict with t_2 if t_3 fires before t_1 as shown in Fig. 10.9(c).

Concurrent transitions t_2 and t_3 in Fig. 10.10(a) model concurrent execution of two processes. A *marked graph* can model concurrency, but not choice; transitions t_2 and t_3 in Fig. 10.10(b) are concurrent; there is no causal relationship between them. Transition t_4 and its input places p_3 and p_4 in Fig. 10.10(b) model synchronization; t_4 fire only if conditions associated with p_3 and p_4 are satisfied.

PNs can be used to model *priorities*. The net in Fig. 10.10(c) models a system with two processes modeled by transitions t_1 and t_2; the process modeled by t_2 has a higher priority than the one modeled by t_1. If both processes are ready to run, places p_1 and p_2 hold tokens. When the two processes are ready, transition t_2 will fire first, modeling the activation of the second process. Only after t_2 is activated, transition t_1, modeling the activation of the first process, will fire.

PNs are able to model *exclusion*, for example, the net in Fig. 10.10(d) models a group of n concurrent processes in a shared-memory environment. At any given time only one process may write, but any subset of the n processes may read at the same time, provided that no process writes. Place p_3 models the process allowed to write, p_4 the ones allowed to read, p_2 the ones ready to access the shared memory, and p_1 the running tasks. Transition t_2 models the initialization/selection of the process allowed to write and t_1 of the processes allowed to read, whereas t_3 models the completion of a write and t_4 the completion of a read. Indeed, p_3 may have at most one token, while p_4 may have at most n. If all n processes are ready to access the shared memory, all n tokens in p_2 are consumed when transition t_1 fires. However, place p_4 may contain n tokens obtained by successive firings of transition t_2.

Formal definitions. After this informal discussion of PNs, we switch to a more formal presentation and give several definitions.

Labeled Petri Net: a tuple $N = (p, t, f, l)$ such that:

- $p \subseteq U$ is a finite set of *places*,
- $t \subseteq U$ is a finite set of *transitions*,
- $f \subseteq (p \times t) \cup (t \times p)$ a set of directed arcs, called *flow relations*,
- $l : t \to L$ a labeling or a weight function

with U a universe of identifiers and L a set of labels. The weight function describes the number of tokens necessary to enable a transition. Labeled PNs describe a static structure; places may contain *tokens* and the distribution of tokens over places defines the state, or the markings of the PN. The dynamic behavior of a PN is described by the structure, together with the markings of the net.

Marked Petri Net: a pair (N, s) where $N = (p, t, f, l)$ is a labeled PN and s is a bag[4] over p denoting the markings of the net.

Preset and Postset of Transitions and Places. The preset of transition t_i, denoted as $\bullet t_i$, is the set of input places of t_i and the postset, denoted by $t_i \bullet$, is the set of the output places of t_i. The preset of place p_j, denoted as $\bullet p_j$, is the set of input transitions of p_j and the postset, denoted by $p_j \bullet$, is the set of the output transitions of p_j.

Fig. 10.8(a) shows a PN with three places, p_1, p_2, and p_3, and one transition, t_1. The weights of the arcs from p_1 and p_2 to t_1 are two and one, respectively; the weight of the arc from t_1 to p_3 is three. The preset of transition t_1 in Fig. 10.8(a) consists of two places, $\bullet t_1 = \{p_1, p_2\}$ and its postset consist of only one place, $t_1 \bullet = \{p_3\}$. The preset of place p_4 in Fig. 10.10(a) consists of transitions t_3 and t_4, $\bullet p_4 = \{t_3, t_4\}$ and the postset of p_1 is $p_1 \bullet = \{t_1, t_2\}$.

Ordinary Net. A PN is ordinary if the weights of all arcs are 1. The nets in Figs. 10.10(a), (b), and (c) are ordinary nets; the weights of all arcs are 1.

Enabled Transition: a transition $t_i \in t$ of the ordinary PN (N, s), with s the initial marking of N, *is enabled* if and only if each of its input places contain a token, $(N, s)[t_i > \Leftrightarrow \bullet t_i \in s$. The notation

[4] A bag $\mathcal{B}(\mathcal{A})$ is a multiset of symbols from an alphabet, $\mathcal{A}$; it is a function from $\mathcal{A}$ to the set of natural numbers. For example, $[x^3, y^4, z^5, w^6 \mid P(x, y, z, w)]$ is a bag consisting of three elements x, four elements y, five elements z, and six elements w such that the $P(x, y, z, w)$ holds. P is a predicate on symbols from the alphabet. x is an element of a bag A denoted as $x \in A$ if $x \in \mathcal{A}$ and if $A(x) > 0$.

$(N, s)[t_i >$ means that t_i is enabled. The marking of a PN changes as a result of transition firing; a transition must be enabled to fire.

Firing Rule: the firing of the transition t_i of the ordinary net (N, s) means that a token is removed from each of its input places, and one token is added to each of its output places, its marking changes $s \mapsto (s - \bullet t_i + t_i \bullet)$. Thus, firing of transition t_i changes a marked net (N, s) into another marked net $(N, s - \bullet t_i + t_i \bullet)$.

Firing Sequence: a nonempty sequence of transitions $\sigma \in t^*$ of the marked net (N, s_0) with $N = (p, t, f, l)$ is called a *firing sequence* if and only if there exist markings $s_1, s_2, \ldots, s_n \in \mathcal{B}(p)$ and transitions $t_1, t_2, \ldots, t_n \in t$ such that $\sigma = t_1, t_2, \ldots, t_n$ and for $i \in (0, n)$, $(N, s_i)[t_{i+1}] >$ and $s_{i+1} = s_i - \bullet t_i + t_i \bullet$. All firing sequences that can be initiated from marking s_0 are denoted as $\sigma(s_0)$.

Reachability is the problem of finding if marking s_n is reachable from the initial marking s_0, with $s_n \in \sigma(s_0)$. Reachability is a fundamental concern for dynamic systems. The reachability problem is decidable; reachability algorithms require exponential time and space.

Liveness: a marked Petri net (N, s_0) is said to be *live* if it is possible to fire any transition infinitely often, starting from the initial marking, s_0. The absence of deadlock in a system is guaranteed by the liveness of its net model.

Incidence Matrix: given a Petri net with n transitions and m places, the incidence matrix $F = [f_{i,j}]$ is an integer matrix with $f_{i,j} = w(i, j) - w(j, k)$. Here, $w(i, j)$ is the weight of the flow relation (arc) from transition t_i to its output place p_j, and $w(j, k)$ is the weight of the arc from the input place p_j to transition t_k. In this expression, $w(i, j)$ represents the number of tokens added to the output place p_j and $w(j, k)$ the number of tokens removed from the input place p_j when transition t_i fires. F^T is the transpose of the incidence matrix.

A marking s_k can be written as a $m \times 1$ column vector, and its j-th entry denotes the number of tokens in place j after some transition firing. The necessary and sufficient condition for transition t_k to be enabled at a marking s is that $w(j, k) \leq s(j) \ \forall s_j \in \bullet t_i$, the weight of the arc from every input place of the transition, be smaller or equal to the number of tokens in the corresponding input place.

Extended Nets: PNs with inhibitor arcs; an *inhibitor arc* prevents the enabling of a transition. For example, the arc from p_2 to t_1 in the net in Fig. 10.10(a) is an inhibitor arc; the process modeled by transition t_1 can be activated only after the process modeled by transition t_2 is activated.

Modified Transition Enabling Rule for Extended Nets: a transition is not enabled if one of the places in its preset is connected with the transition with an inhibitor arc and if the place holds a token. For example, transition t_1 in the net in Fig. 10.10(c) is not enabled while place p_2 holds a token.

There are several classes of PNs distinguished from one another by their structural properties:

1. State Machines—are used to model finite state machines and cannot model concurrency and synchronization.
2. Marked Graphs—cannot model choice and conflict.
3. Free-choice Nets—cannot model confusion.
4. Extended Free-choice Nets—cannot model confusion but allow inhibitor arcs.
5. Asymmetric Choice Nets—can model asymmetric confusion but not symmetric ones.

This partitioning is based on the number of input and output flow relations from/to a transition or a place and by the manner in which transitions share input places, as indicated in Fig. 10.11.

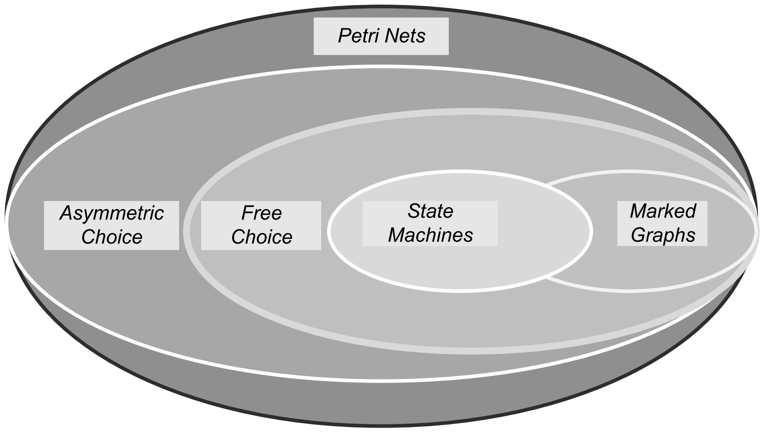

FIGURE 10.11

Classes of Petri nets.

State Machine: a PN is a state machine if and only if

$$\mid \bullet t_i \mid = 1 \wedge \mid t_i \bullet \mid = 1, \forall t_i \in t. \tag{10.2}$$

All transitions of a state machine have exactly one incoming and one outgoing arc. This topological constraint limits the expressiveness of a state machine; no concurrency is possible. For example, the transitions t_1, t_2, t_3, and t_4 of state machine in Fig. 10.10(a) have only one input and one output arc; the cardinality of their presets and postsets is one. No concurrency is possible; once a choice was made by firing either t_1, or t_2, the evolution of the system is entirely determined. This state machine has four places p_1, p_2, p_3, and p_4, and the marking is a 4-tuple (p_1, p_2, p_3, p_4); the possible markings of this net are $(1, 0, 0, 0)$, $(0, 1, 0, 0)$, $(0, 0, 1, 0)$, $(0, 0, 0, 1)$, with a token in places p_1, p_2, p_3, or p_4, respectively.

Marked Graph: a PN is a marked graph if and only if

$$\mid \bullet p_i \mid = 1 \wedge \mid p_i \bullet \mid = 1, \forall p_i \in p. \tag{10.3}$$

In a marked graph, each place has only one incoming and one outgoing flow relation; thus, marked graphs do no not allow modeling of choice.

Free Choice, Extended Free Choice, and Asymmetric Choice Petri Nets: the marked net, (N, s_0) with $N = (p, t, f, l)$ is a free-choice net if and only if

$$(\bullet t_i) \cap (\bullet t_j) = \emptyset \Rightarrow \mid \bullet t_i \mid = \mid \bullet t_j \mid, \quad \forall t_{i,j} \in t. \tag{10.4}$$

N is an extended free-choice net if $(\bullet t_i) \cap (\bullet t_j) = \emptyset \Rightarrow \bullet t_i = \bullet t_j, \forall t_i, t_j \in t$.

N is an asymmetric choice net if and only if $(\bullet t_i) \cap (\bullet t_j) \neq \emptyset \Rightarrow (\bullet t_i \subseteq \bullet t_j)$ or $(\bullet t_i \supseteq \bullet t_j), \forall t_i, t_j \in t$.

In an extended free-choice net, if two transitions share an input place, they must share all places in their presets. In an asymmetric choice net two transitions may share only a subset of their input places.

Several extensions of PNs have been proposed. For example, Colored Petri Nets (CPNs) allow tokens of different colors, thus increasing the expressivity of the PNs but not simplifying their analysis. Several extensions of PNs to support performance analysis by associating a random time with each transition have been proposed. In case of Stochastic Petri Nets (SPNs), a random time elapses between the time a transition is enabled and the moment it fires. This random time allows the model to capture the service time associated with the activity modeled by the transition.

Applications of SPNs to performance analysis of complex systems is generally limited by the explosion of the state space of the models. Stochastic High-Level Petri Nets (SHLPN) were introduced in 1988 [309]; the SHLPNs allow easy identification of classes of equivalent markings even when the corresponding aggregation of states in the Markov domain is not obvious. This aggregation could reduce the size of the state space by one or more orders of magnitude depending on the system being modeled.

Cloud applications often require a large number of tasks to run concurrently. The interdependencies among these tasks are quite intricate, and PNs can be very useful to intuitively present a high-level model of the interactions among the tasks. The five classes of systems mentioned earlier, including finite-state machines, as well as control flow and data flow systems, can be modeled by PNs. The workflow patterns common to many cloud applications discussed in Section 11.2 translate easily to PN models. Unfortunately, the size of the state space of detailed PN models of complex systems grows very fast, and this limits the usefulness of these models.

10.7 Process state; global state of a process or thread group

To understand the important properties of distributed systems, we use a model, an abstraction based on two critical components, processes/threads and communication channels. A *process* is a program in execution, and a *thread* is a light-weight process. A thread of execution is the smallest unit of processing that can be scheduled by an operating system.

A process or a thread is characterized by its *state*. The state is the ensemble of information that we need to restart a process or thread after it was suspended. An *event* is a change of state of a process or a thread. The events affecting the state of process p_i are numbered sequentially as $e_i^1, e_i^2, e_i^3, \ldots$, as shown in the space-time diagram in Fig. 10.12(a). A process p_i is in state σ_i^j immediately after the occurrence of event e_i^j and remains in that state until the occurrence of the next event, e_i^{j+1}.

A process or a thread group is a collection of cooperating processes and threads; to reach a common goal, the processes work in concert and communicate with one another. For example, a parallel algorithm to solve a system of partial deferential equations (PDEs) over a domain D may partition the data in several segments and assign each segment to one of the members of the group. The processes or the treads in the group must cooperate with one another and iterate until the common boundary values computed by one process agree with the common boundary values computed by another.

A *communication channel* provides the means for processes/threads to communicate with one another and coordinate their actions by exchanging messages. Without loss of generality, we assume that

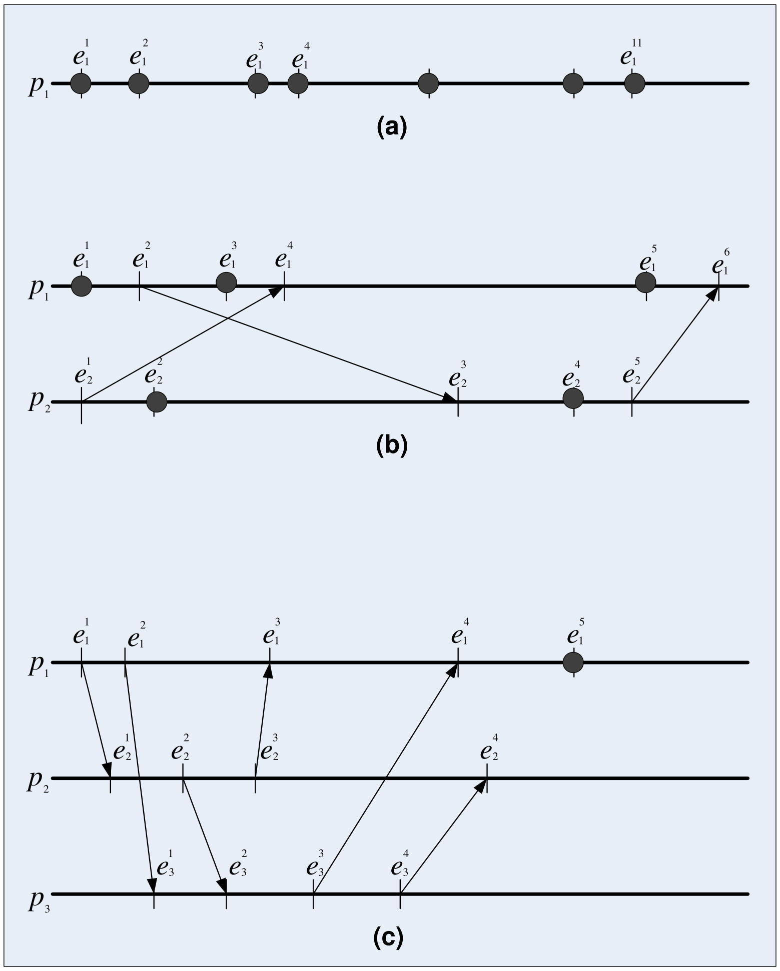

FIGURE 10.12

Space-time diagrams display local and communication events during a process lifetime. Local events are small black circles. Communication events in different processes/threads are connected by lines originating at a *send* event and terminated by an arrow at the *receive* event. (a) All events in the case of a single process p_1 are local; the process is in state σ_1 immediately after the occurrence of event e_1^1 and remains in that state until the occurrence of event e_1^2. (b) Two processes/threads p_1 and p_2; event e_1^2 is a communication event; p_1 sends a message to p_2; event e_2^3 is a communication event, process or thread p_2 receives the message sent by p_1. (c) Three processes or threads interact by means of communication events.

communication among processes is done only by means of *send(m)* and *receive(m)* communication events, where *m* is a message. We use the term “message” for a structured unit of information that can be interpreted only in a semantic context by the sender and the receiver. The *state of a communication* *channel* is defined as follows: given two processes p_i and p_j, the state of the channel, $\xi_{i,j}$, from p_i to p_j consists of messages sent by p_i but not yet received by p_j.

These two abstractions allow us to concentrate on critical properties of distributed systems without the need to discuss the detailed physical properties of the entities involved. The model presented is based on the assumption that a channel is a unidirectional bit pipe of infinite bandwidth and zero latency, but unreliable; messages sent through a channel may be lost or distorted or the channel may fail, losing its ability to deliver messages. We also assume that the time a process needs to traverse a set of states is of no concern and that processes may fail, or be aborted.

A *protocol* is a finite set of messages exchanged among processes and threads to help them coordinate their actions. Fig. 10.12(c) illustrates the case when communication events are dominant in the local history of processes, p_1, p_2, and p_3. In this case, only e_1^5 is a local event; all others are communication events. The particular protocol illustrated in Fig. 10.12(c) requires processes p_2 and p_3 to send messages to the other processes in response to a message from process p_1.

Informal definition of the state of a single process or a thread can be extended to collections of communicating processes/threads. The *global state of a distributed system* consisting of several processes and communication channels is the union of the states of individual processes and channels [40].

Call h_i^j the history of process p_i up to and including its j-th event, e_i^j, and call σ_i^j the local state of process p_i following event e_i^j. Consider a system consisting of n processes, $p_1, p_2, \ldots, p_i, \ldots, p_n$ with $\sigma_i^{j_i}$ the local state of process p_i; then, the global state of the system is an n-tuple of local states

$$\Sigma^{(j_1,j_2,\ldots,j_n)} = (\sigma_1^{j_1}, \sigma_2^{j_2}, \ldots, \sigma_i^{j_i}, \ldots, \sigma_n^{j_n}). \tag{10.5}$$

The state of the channels does not appear explicitly in this definition of the global state because the state of the channels is encoded as part of the local state of the processes/threads communicating through the channels. The global states of a distributed computation with n processes form an n-dimensional lattice. The elements of this lattice are global states $\Sigma^{(j_1,j_2,\ldots,j_n)}(\sigma_1^{j_1}, \sigma_2^{j_2}, \ldots, \sigma_n^{j_n})$.

Fig. 10.13(a) shows the lattice of global states of the distributed computation in Fig. 10.12(b). This is a two-dimensional lattice because we have two processes, p_1 and p_2. The lattice of global states for the distributed computation in Fig. 10.12(c) is a three-dimensional lattice; the computation consists of three concurrent processes, p_1, p_2, and p_3.

The initial state of the system in Fig. 10.13(b) is the state before the occurrence of any event, and it is denoted by $\Sigma^{(0,0)}$; the only global states reachable from $\Sigma^{(0,0)}$ are $\Sigma^{(1,0)}$, and $\Sigma^{(0,1)}$. The communication events limit the global states the system may reach; in this example the system cannot reach the state $\Sigma^{(4,0)}$ because process p_1 enters state σ_4 only after process p_2 has entered the state σ_1. Fig. 10.13(b) shows the six possible sequences of events to reach the global state $\Sigma^{(2,2)}$:

$$(e_1^1, e_1^2, e_2^1, e_2^2), (e_1^1, e_2^1, e_1^2, e_2^2), (e_1^1, e_2^1, e_2^2, e_1^2), (e_2^1, e_2^2, e_1^1, e_1^2), (e_2^1, e_1^1, e_1^2, e_2^2), (e_2^1, e_1^1, e_2^2, e_1^2). \tag{10.6}$$

An interesting question is: How many paths needed to reach a global state exist? The more paths exist, the harder it is to identify the events leading to a state when we observe an undesirable behavior of the system. A large number of paths increases the difficulties to debug the system.

We conjecture that, in the case of two threads in Fig. 10.13(a), the number of paths from the global state $\Sigma^{(0,0)}$ to $\Sigma^{(m,n)}$ is

$$N_p^{(m,n)} = \frac{(m+n)!}{m!n!}. \tag{10.7}$$

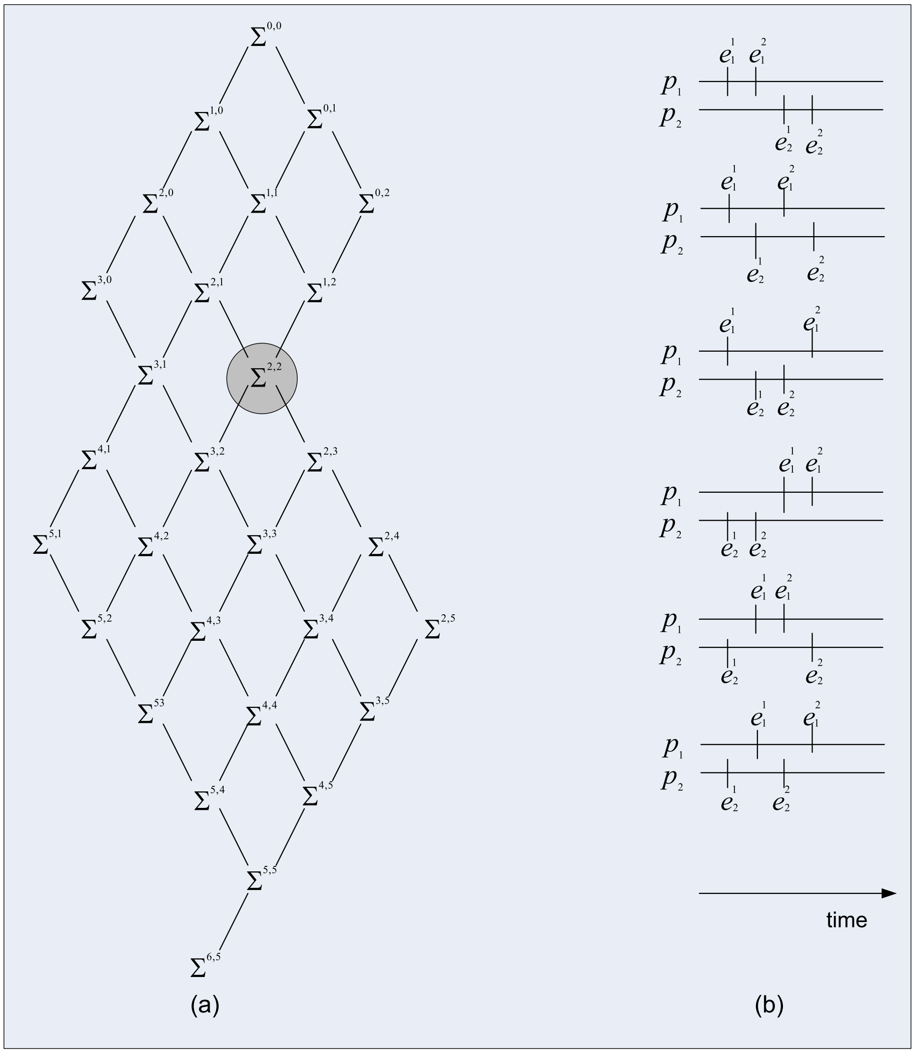

FIGURE 10.13

(a) The lattice of the global states of two processes/threads with the space–time diagrams in Fig. 10.12(b). (b) The six possible sequences of events leading to the state $\Sigma^{(2,2)}$.

We have already seen that there are six paths leading to state $\Sigma^{(2,2)}$ and, indeed

$$N_p^{(2,2)} = \frac{(2+2)!}{2!2!} = \frac{24}{4} = 6. \tag{10.8}$$

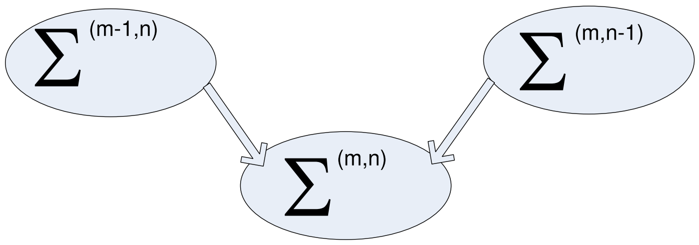

FIGURE 10.14

In the two-dimensional case the global state $\Sigma^{(m,n)}$, $\forall(m,n) \geq 1$ can only be reached from two states, $\Sigma^{(m-1,n)}$ and $\Sigma^{(m,n-1)}$.

To prove Eq. (10.7), we use a method resembling induction; we notice first that the global state $\Sigma^{(1,1)}$ can only be reached from the states $\Sigma^{(1,0)}$ and $\Sigma^{(0,1)}$ and that $N_p^{(1,1)} = (2)!/1!1! = 2$; thus the formula is true for $m = n = 1$. Then, we show that, if the formula is true for the $(m-1, n-1)$ case, it will also be true for the (m, n) case. If our conjecture is true, then

$$N_p^{[(m-1),n]} = \frac{[(m-1)+n)]!}{(m-1)!n!} \tag{10.9}$$

and

$$N_p^{[m,(n-1)]} = \frac{[(m+(n-1)]!}{m!(n-1)!}. \tag{10.10}$$

We observe that the global state $\Sigma^{(m,n)}$, $\forall(m,n) \geq 1$ can only be reached from two states, $\Sigma^{(m-1,n)}$ and $\Sigma^{(m,n-1)}$, as seen in Fig. 10.14, thus

$$N_p^{(m,n)} = N_p^{(m-1,n)} + N_p^{(m,n-1)}. \tag{10.11}$$

Indeed

$$\frac{[(m-1)+n]!}{(m-1)!n!} + \frac{[m+(n-1)]!}{m!(n-1)!} = (m+n-1)!\left[\frac{1}{(m-1)!n!} + \frac{1}{m!(n-1)!}\right] = \frac{(m+n)!}{m!n!}. \tag{10.12}$$

This shows that our conjecture is true, and, thus, Eq. (10.7) gives the number of paths to reach the global state $\Sigma^{(m,n)}$ from $\Sigma^{(0,0)}$ when two threads are involved. This expression can be generalized for the case of q threads; using the same strategy, we see that the number of paths from the state $\Sigma^{(0,0,\ldots,0)}$ to the global state $\Sigma^{(n_1,n_2,\ldots,n_q)}$ is

$$N_p^{(n_1,n_2,\ldots,n_q)} = \frac{(n_1+n_2+\ldots+n_q)!}{n_1!n_2!\ldots n_q!}. \tag{10.13}$$

Indeed,

$$N_p^{(n_1,n_2,\ldots,n_q)} = N_p^{(n_1-1,n_2,\ldots,n_q)} + N_p^{(n_1,n_2-1,\ldots,n_q)} + \ldots + N_p^{(n_1,n_2,\ldots,n_q-1)}. \tag{10.14}$$

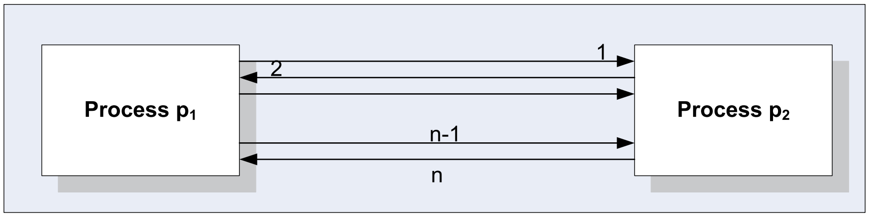

FIGURE 10.15

Process coordination in the presence of errors; each message may be lost with probability ϵ. If a protocol consisting of n messages exists, then the protocol should be able to function properly with $n-1$ messages reaching their destination, one of them being lost.

Eq. (10.13) gives us an indication of how difficult it is to debug a system with a large number of concurrent threads.

Many problems in distributed systems are instances of the *global predicate evaluation problem* (GPE) where the goal is to evaluate a Boolean expression whose elements are functions of the global state of the system.

10.8 Communication protocols and process coordination

A major concern in any parallel and distributed system is communication in the presence of channel failures. There are multiple modes for a channel to fail, and some lead to messages being lost. In the general case, it is impossible to guarantee that two processes will reach an agreement in case of channel failures; see Fig. 10.15.

Statement. Given two processes p_1 and p_2 connected by a communication channel that can lose a message with probability $\epsilon > 0$, no protocol capable of guaranteeing that two processes will reach agreement exists, regardless of how small the probability ϵ is.

The proof of this statement is by contradiction. Assume that such a protocol exists and it consists of n messages. Recall that a protocol consists of a finite number of messages. Since any message might be lost with probability ϵ, the protocol should be able to function when only $n-1$ messages reach their destination, the last one being lost. Induction on the number of messages proves that indeed no such protocol exists; indeed, the same reasoning leads us to conclude that the protocol should function correctly with $n-2$ messages, and so on.

In practice, error detection and error correction codes allow processes to communicate reliably through noisy digital channels. The redundancy of a message is increased by more bits and packaging the message as a codeword; the recipient of the message is then able to decide if the sequence of bits received is a valid codeword, and, if the code satisfies some distance properties, then the recipient of the message is able to extract the original message from a bit string in error.

Communication protocols implement not only *error control* mechanisms, but also flow control and congestion control. *Flow control* provides feedback from the receiver; it forces the sender to transmit only the amount of data the receiver is able to buffer and then process. *Congestion control* ensures that the offered load of the network does not exceed the network capacity. In store-and-forward networks, individual routers may drop packets when the network is congested and the sender is forced to retransmit. Based on the estimation of the RTT (Round-Trip-Time), the sender can detect congestion and reduce the transmission rate.

The implementation of these mechanisms requires the measurement of *time intervals*, the time elapsed between two events; we also need a *global concept of time* shared by all entities that cooperate with one another. For example, a computer chip has an *internal clock* and a predefined set of actions occurs at each clock tick. Each chip has an *interval timer* that helps enhance the system's fault tolerance; when the effects of an action are not sensed after a predefined interval, the action is repeated.

When the entities communicating with each other are networked computers, clock synchronization precision is critical [291]. The event rates are very high, each system goes through state changes at a very fast pace; modern processors run at a 2–4 GHz clock rate. This explains why we need to measure time very accurately; indeed, we have atomic clocks with an accuracy of about 10^{-6} seconds per year.

An isolated system can be characterized by its *history* expressed as a sequence of events, each event corresponding to a change of the state of the system. Local timers provide relative time measurements. A more accurate description adds to the system's history the time of occurrence of each event as measured by the local timer.

Determining the global state of a large-scale distributed system is a very challenging problem. Messages sent by processes may be lost or distorted during transmission. There are no means to ensure a perfect synchronization of local clocks and no obvious methods to ensure a global ordering of events occurring in different processes unless we restricting message delays and the type of errors.

The mechanisms just described are insufficient once we approach the problem of cooperating entities. To coordinate their actions, two entities need a common perception of time. Timers are not enough; clocks provide the only way to measure distributed duration, that is, actions that start in one process and terminate in another.

Global agreement on time is necessary to trigger actions that should occur concurrently. For example, in a real-time control system of a power plant, several circuits must be switched on at the same time. Agreement on *the time when events occur* is necessary for a distributed recording of events, for example, to determine a precedence relationship through a temporal ordering of events. To ensure that a system functions correctly, we need to determine that the event causing a change of state occurred before the state change, e.g., the sensor triggering an alarm has to change its value before the emergency procedure to handle the event was activated. Another example of the need for agreement on the time of occurrence of events is in replicated actions. In this case, several replicas of a process must log the time of an event in a consistent manner.

Time stamps are often used for event ordering using a global time-base constructed on local virtual clocks [338]. The Δ-protocols [119] achieve total temporal order using a global time base. Assume that local virtual clock readings do not differ by more than π, called *precision* of the global time base. Call g the *granularity of physical clocks*. First, observe that the granularity should not be smaller than the precision; given two events a and b occurring in different processes, if $t_b - t_a \leq \pi + g$, we cannot tell which of a or b occurred first [493]. Based on these observations, it follows that the order discrimination of clock-driven protocols cannot be better than twice the clock granularity.

System specification, design, and analysis require a clear understanding of *cause–effect relationships*. During the system specification phase, we view the system as a state machine and define the actions that cause transitions from one state to another. During the system analysis phase, we need to determine the cause that brought the system to a certain state.

The activity of any process is modeled as a sequence of *events*; hence, the binary relationship cause–effect should be expressed in terms of events and should support our intuition that *the cause must precede the effect*. Again, we need to distinguish between local events and communication events. The latter ones affect more than one process and are essential for constructing a global history of an ensemble of processes. Let h_i denote the local history of process p_i, and let e_i^k denote the k-th event in this history.

The binary cause–effect relationship between two events has the following properties:

1. Causality of local events can be derived from the process history. Given two events e_i^k and e_i^l local to process $p_{i,}$

$$\text{if } e_i^k, e_i^l \in h_i \text{ and } k < l, \text{ then } e_i^k \rightarrow e_i^l. \tag{10.15}$$

2. Causality of communication events. Given two processes p_i and p_j and two events e_i^k and e_j^l,

$$\text{if } e_i^k = send(m) \text{ and } e_j^l = receive(m), \text{ then } e_i^k \rightarrow e_j^l. \tag{10.16}$$

3. Transitivity of the causal relationship. Given three processes, p_i, p_j, and p_m and the events e_i^k, e_j^l, and e_m^n occurring in p_i, p_j, and p_m, respectively,

$$\text{if } e_i^k \rightarrow e_j^l \text{ and } e_j^l \rightarrow e_m^n, \text{ then } e_i^k \rightarrow e_m^n. \tag{10.17}$$

Two events in the global history may be unrelated, neither one being the cause of the other; such events are said to be *concurrent events*.

10.9 Communication, logical clocks, and message delivery rules

We need to bridge the gap between the physical systems and the abstractions used to describe interacting processes. This section addresses the means to bridge this gap. Communicating processes often run on distant systems whose physical clocks cannot be perfectly synchronized due to communication latency. Global ordering of events in communicating processes running on such systems is not feasible, and logical clocks are used instead. Also, messages travel through physical channels with different speeds and follow different paths. As a result, the order in which messages are delivered to processes may be different than the order they were sent.

Logical clocks. A *logical clock (LC)* is an abstraction necessary to ensure the clock condition given by Eqs. (10.24) and (10.25) in the absence of a global clock. Each process p_i maps events to positive integers. Call $LC(e)$ the local variable associated with event e. Each process time stamps each message m sent with the value of the logical clock at the time of sending, $TS(m) = LC(send(m))$. The rules to update the logical clock are specified by the following relationship:

$$LC(e) := \begin{cases} LC + 1 & \text{if } e \text{ is a local event or a } send(m) \text{ event} \\ \max(LC, TS(m) + 1) & \text{if } e = receive(m). \end{cases} \tag{10.18}$$

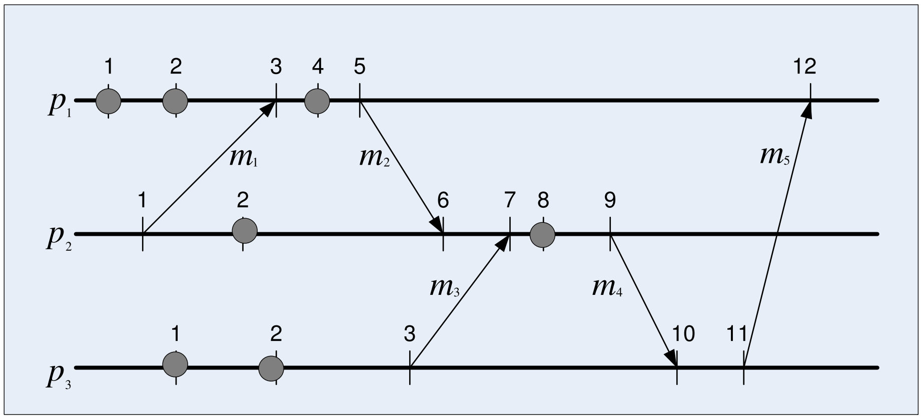

FIGURE 10.16

Three processes and their logical clocks; The usual labeling of events as $e_1^1, e_1^2, e_1^3, \ldots$ is omitted to avoid overloading the figure; only the logical clock values for the local and for the communication events are marked. The correspondence between the events and the logical clock values is obvious: $e_1^1, e_2^1, e_3^1 \rightarrow 1$, $e_1^5 \rightarrow 5$, $e_2^4 \rightarrow 7$, $e_3^4 \rightarrow 10$, $e_1^6 \rightarrow 12$, etc. Global ordering of all events is not possible; there is no way to establish the ordering of events e_1^1, e_2^1, and e_3^1.

The concept of logical clocks is illustrated in Fig. 10.16 using a modified *space–time diagram*, where the events are labeled with the logical clock value. Messages exchanged between processes are shown as lines from the sender to the receiver; the communication events corresponding to sending and receiving messages are marked on these diagrams.

Each process labels local events and send events sequentially until it receives a message marked with a logical clock value larger than the next local logical clock value, as shown in Eq. (10.18). It follows that logical clocks do not allow a global ordering of all events. For example, there is no way to establish the ordering of events e_1^1, e_2^1, and e_3^1 in Fig. 10.16. Nevertheless, communication events allow different processes to coordinate their logical clocks; for example, process p_2 labels the event e_2^3 as 6 because of message m_2, which carries the information about the logical clock value as 5 at the time message m_2 was sent. Recall that e_i^j is the j-th event in process p_i.

Logical clocks lack an important property, *gap detection*; given two events e and e' and their logical clock values, $LC(e)$ and $LC(e')$, it is impossible to establish if an event e'' exists such that

$$LC(e) < LC(e'') < LC(e'). \tag{10.19}$$

For example, for process p_1, there is an event, e_1^4, between the events e_1^3 and e_1^5 in Fig. 10.16; indeed, $LC(e_1^3) = 3$, $LC(e_1^5) = 5$, $LC(e_1^4) = 4$, and $LC(e_1^3) < LC(e_1^4) < LC(e_1^5)$. However, for process p_3, the events e_3^3 and e_3^4 are consecutive though $LC(e_3^3) = 3$ and $LC(e_3^4) = 10$.

Message delivery rules. The communication channel abstraction makes no assumptions about the order of messages; a real-life network might reorder messages. This fact has profound implications

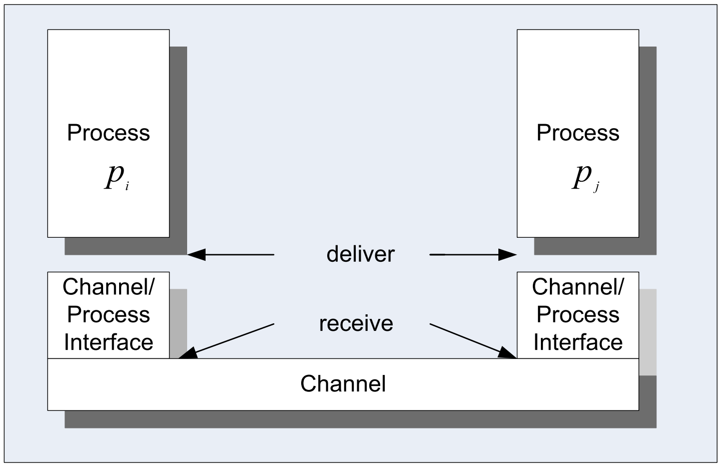

FIGURE 10.17

Message receiving and message delivery are two distinct operations. The channel-process interface implements the delivery rules, e.g., FIFO delivery.

for a distributed application. Consider, for example, a robot getting instructions to navigate from a monitoring facility with two messages, "turn left" and "turn right", being delivered out of order.

Message receiving and message delivery are two distinct operations; a *delivery rule* is an additional assumption about the channel-process interface. This rule establishes when a message received is actually delivered to the destination process. The receiving of a message m and its delivery are two distinct events in a causal relationship with one another: A message can only be delivered after being received; see Fig. 10.17:

$$receive(m) \rightarrow deliver(m). \tag{10.20}$$

First-In-First-Out (FIFO) delivery implies that messages are delivered in the same order they are sent. For each pair of source-destination processes (p_i, p_j), FIFO delivery requires that the following relationship should be satisfied:

$$send_i(m) \rightarrow send_i(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m'). \tag{10.21}$$

Even if the communication channel does not guarantee FIFO delivery, FIFO delivery can be enforced by attaching a sequence number to each message sent. The sequence numbers are also used to reassemble messages out of individual packets.

Causal message delivery. Causal delivery is an extension of the FIFO delivery to the case when a process receives messages from different sources. Assume a group of three processes (p_i, p_j, p_k) and two messages m and m'. Causal delivery requires that

$$send_i(m) \rightarrow send_j(m') \Rightarrow deliver_k(m) \rightarrow deliver_k(m'). \tag{10.22}$$

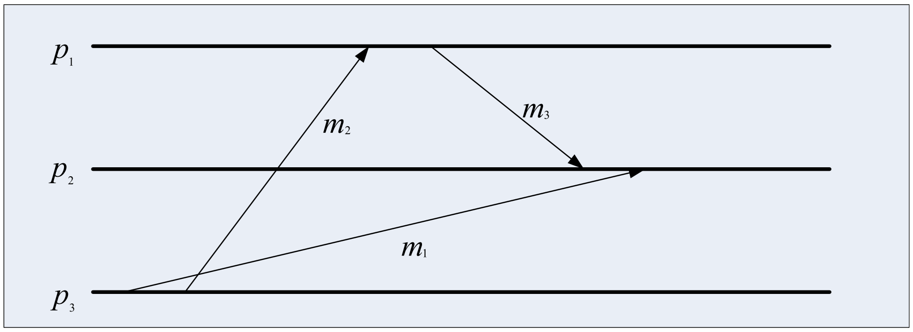

FIGURE 10.18

Violation of causal delivery when more than two processes are involved; message m_1 is delivered to process p_2 after message m_3, though message m_1 was sent before m_3. Indeed, message m_3 was sent by process p_1 after receiving m_2, which in turn was sent by process p_3 after sending message m_1.

When more than two processes are involved in a message exchange, the message delivery may be FIFO, but not causal as shown in Fig. 10.18 where we see that

- $deliver(m_3) \rightarrow deliver(m_1)$; according to the local history of process p_2.
- $deliver(m_2) \rightarrow send(m_3)$; according to the local history of process p_1.
- $send(m_1) \rightarrow send(m_2)$; according to the local history of process p_3.
- $send(m_2) \rightarrow deliver(m_2)$.
- $send(m_3) \rightarrow deliver(m_3)$.

The transitivity property and the causality relationship imply that

$$send(m_1) \rightarrow deliver(m_3). \tag{10.23}$$

Call $TS(m)$ the *time stamp* carried by message m. A message received by process p_i is *stable* if no future messages with a time stamp smaller than $TS(m)$ can be received by process p_i. When using logical clocks, a process p_i can construct consistent observations of the system if it implements the following delivery rule: *deliver all stable messages in increasing time stamp order.*

Let us now examine the problem of *consistent message delivery* under several sets of assumptions. First, assume that processes cooperating with each other in a distributed environment have access to a *global real-time clock*, that the message delays are bounded by δ, and that there is no clock drift. Call $RC(e)$ the time of occurrence of event e. A process includes in every message it sends the $RC(e)$, where e is the send message event. The delivery rule in this case is: *at time t deliver all received messages with time stamps up to $(t - \delta)$ in increasing time stamp order.* Indeed, this delivery rule guarantees that, under the bounded delay assumption, the message delivery is consistent. All messages delivered at time t are in order, and no future message with a time stamp lower than any of the messages delivered may arrive.

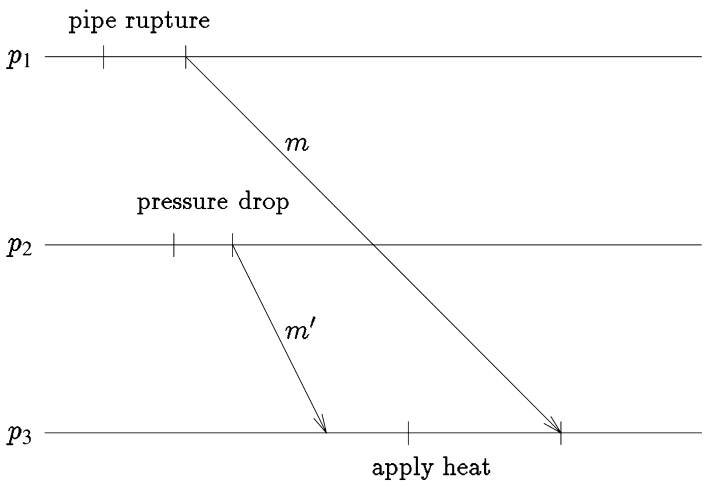

FIGURE 10.19

The causal sequence of events observed by the controller process p_3, namely, pressure drop → apply heat → pipe rupture leads to the erroneous conclusion that the cause of the pipe rupture is the increased temperature.

For any two events, e and e', occurring in different processes, the so-called *clock condition* is satisfied if

$$e \rightarrow e' \Rightarrow RC(e) < RC(e'),\ \forall e, e'. \tag{10.24}$$

Oftentimes, we are interested in determining the set of events that caused an event knowing the time stamps associated with all events; in other words, we are interested in deducing the causal precedence relation between events from their time stamps. To do so, we need to define the so-called *strong clock condition.* The strong clock condition requires an equivalence between the causal precedence and the ordering of the time stamps

$$\forall e, e', \quad e \rightarrow e' \equiv TS(e) < TS(e'). \tag{10.25}$$

Causal delivery is very important because it allows processes to reason about the entire system using only local information. This is only true in a closed system where all communication channels are known. Sometimes, a system has a *hidden channel* and reasoning based on causal analysis may lead to incorrect conclusions [40]. This is the case of the example in Fig. 10.19 in which the environment acts as a hidden channel.

In this example process p_1 detects the rupture of a steam pipe and sends message m to process p_3. Process p_2 monitors the pipe pressure, and a few seconds after the rupture of the pipe detects a drop in the pressure and sends message m' to p_3. Messages m and m' are concurrent from the point of view of explicit communication. p_3, the controller process, reacts to the pressure drop by applying more heat to increase the temperature. Message m reporting the rupture of the pipe arrives moments later. The causal sequence of events observed by the controller process, pressure drop, apply heat, pipe rupture, leads to the erroneous conclusion that the pipe rupture is due to increased temperature [281]!

10.10 Runs and cuts; causal history

We often need to know the state of several, possibly all, processes of a distributed system. For example, a supervisory process must be able to detect when a subset of processes is deadlocked; a process might migrate from one location to another or be replicated only after an agreement with others. In all these examples, a process needs to evaluate a predicate function of the global state of the system.

We call the process responsible for constructing the global state of the system the *monitor*; a monitor sends messages requesting information about the local state of every process and gathers the replies to construct the global state. Intuitively, the construction of the global state is equivalent to taking snapshots of individual processes and then combining these snapshots into a global view. Yet, combining snapshots is straightforward if and only if all processes have access to a global clock and the snapshots are taken at the same time; hence, the snapshots are contemporaneous with one another.

A *run* is a total ordering R of all the events in the global history of a distributed computation consistent with the local history of each participant process; a run

$$R = (e_1^{j_1}, e_2^{j_2}, \dots, e_n^{j_n}) \tag{10.26}$$

implies a sequence of events, as well as a sequence of global states.

For example, consider the three processes in Fig. 10.20. We can construct a three-dimensional lattice of global states following a procedure similar to the one in Fig. 10.13, starting from the initial state $\Sigma^{(000)}$ and proceeding to any reachable state $\Sigma^{(ijk)}$ with i, j, k being the events in processes p_1, p_2, p_3, respectively. The run $R_1 = (e_1^1, e_2^1, e_3^1, e_1^2)$ is consistent with both the local history of each process and the global history; this run is valid, and the system has traversed the global states

$$\Sigma^{000}, \Sigma^{100}, \Sigma^{110}, \Sigma^{111}, \Sigma^{211}. \tag{10.27}$$

On the other hand, the run $R_2 = (e_1^1, e_1^2, e_3^1, e_1^3, e_3^2)$ is invalid because it is inconsistent with the global history. The system cannot ever reach the state Σ^{301}; message m_1 must be sent before it is received, so event e_2^1 must occur in any run before event e_1^3.

A *cut* is a subset of the local history of all processes. If h_i^j denotes the history of process p_i up to and including its j-th event, e_i^j, then a cut C is an n-tuple

$$C = \{h_i^j\} \quad \text{with} \quad i \in \{1, n\} \text{ and } j \in \{1, n_i\}. \tag{10.28}$$

The frontier of the cut is an n-tuple consisting of the last event of every process included in the cut. Fig. 10.20 illustrates a *space–time diagram* for a group of three processes, p_1, p_2, p_3, and it shows two cuts, C_1 and C_2. C_1 has the frontier $(4, 5, 2)$ frozen after the fourth event of process p_1, the fifth event of process p_2 and the second event of process p_3, and C_2 has the frontier $(5, 6, 3)$.

Cuts support the intuition to generate global states based on an exchange of messages between a monitor and a group of processes. A cut represents the instance when requests to report individual state are received by the members of the group. Clearly, not all cuts are meaningful. For example, the cut C_1 with the frontier $(4, 5, 2)$ in Fig. 10.20 violates our intuition regarding causality; it includes e_2^4, the event triggered by the arrival of message m_3 at process p_2 but does not include e_3^3, the event triggered

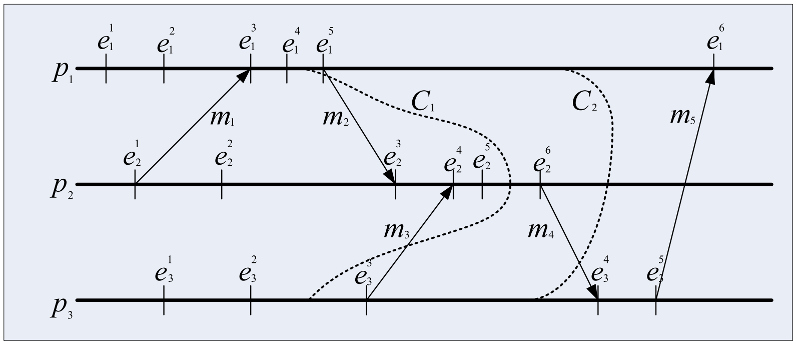

FIGURE 10.20

Inconsistent and consistent cuts: The cut $C_1 = (e_1^4, e_2^5, e_3^2)$ is inconsistent because it includes e_2^4, the event triggered by the arrival of the message m_3 at process p_2, but does not include e_3^3, the event triggered by process p_3 sending m_3, thus the cut C_1 violates causality. On the other hand, $C_2 = (e_1^5, e_2^6, e_3^3)$ is a consistent cut; there is no causal inconsistency; it includes event e_2^6, the sending of message m_4, without the effect of it, the event e_3^4 receiving the message by process p_3.

by process p_3 sending m_3. In this snapshot, p_3 was frozen after its second event, e_3^2, before it had the chance to send message m_3. Causality is violated, and the system cannot ever reach such a state.

Next, we introduce the concepts of consistent and inconsistent cuts and runs. A cut closed under the *causal precedence relationship* is a *consistent cut*. C is a consistent cut if and only if for all events:

$$\forall e, e', \quad (e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C. \tag{10.29}$$

A consistent cut establishes an “instance” of a distributed computation; given a consistent cut, we can determine if an event e occurred before the cut. A run R is said to be consistent if the total ordering of events imposed by the run is consistent with the partial order imposed by the causal relation; for all events, $e \rightarrow e'$ implies that e appears before e' in R.

Consider a distributed computation consisting of a group $G = \{p_1, p_2, ..., p_n\}$ of communicating processes. The *causal history of event e, $\gamma(e)$,* is the smallest consistent cut of G including event e:

$$\gamma(e) = \{e' \in G \mid e' \rightarrow e\} \cup \{e\}. \tag{10.30}$$

The causal history of event e_2^5 in Fig. 10.21 is:

$$\gamma(e_2^5) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_1^5, e_2^1, e_2^2, e_2^3, e_2^4, e_2^5, e_3^1, e_3^3, e_3^3\}. \tag{10.31}$$

This is the smallest consistent cut including e_2^5. Indeed, if we omit e_3^3, then the cut $(5, 5, 2)$ would be inconsistent because it would include e_2^4, the communication event for receiving m_3, but not e_3^3, the event caused by sending m_3. If we omit e_1^5, the cut $(4, 5, 3)$ would also be inconsistent; it would include e_2^3 but not e_1^5.

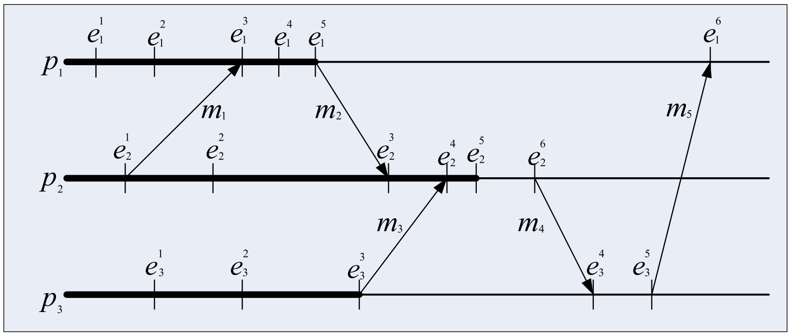

FIGURE 10.21

Causal history of event e_2^5; $\gamma(2^5) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_1^5, e_2^1, e_2^2, e_2^3, e_2^4, e_2^5, e_3^1, e_3^2, e_3^3\}$ is the smallest consistent cut including e_2^5.

Causal histories can be used as clock values and satisfy the strong clock condition, provided that we equate clock comparison with set inclusion. Indeed,

$$e \rightarrow e' \;\equiv\; \gamma(e) \subset \gamma(e'). \tag{10.32}$$

The following algorithm can be used to construct causal histories:

- Each $p_i \in G$ starts with $\theta = \emptyset$.
- Every time p_i receives a message m from p_j it constructs

$$\gamma(e_i) \;=\; \gamma(e_j) \cup \gamma(e_k), \tag{10.33}$$

with e_i the *receive* event, e_j the previous local event of p_i, and e_k the *send* event of process p_j.

Unfortunately, this concatenation of histories is impractical because the causal histories grow very fast.

Now, we present a protocol to construct consistent global states based on the monitoring concepts discusses in this section. We assume a fully connected network. Recall that given two processes p_i and p_j, the state $\xi_{i,j}$ of the channel from p_i to p_j consists of messages sent by p_i but not yet received by p_j. The snapshot protocol of Chandy and Lamport consists of three steps [93]:

1. Process p_0 sends to itself a "take snapshot" message.
2. Let p_f be the process from which p_i receives the "take snapshot" message for the first time. Upon receiving the message, the process p_i records its local state, σ_i, and relays the "take snapshot" along all its outgoing channels without executing any events on behalf of its underlying computation; channel state $\xi_{f,i}$ is set to empty, and process p_i starts recording messages received over each of its incoming channels.
3. Let p_s be the process from which p_i receives the "take snapshot" message beyond the first time; process p_i stops recording messages along the incoming channel from p_s and declares channel state $\xi_{s,i}$ as those messages that have been recorded.

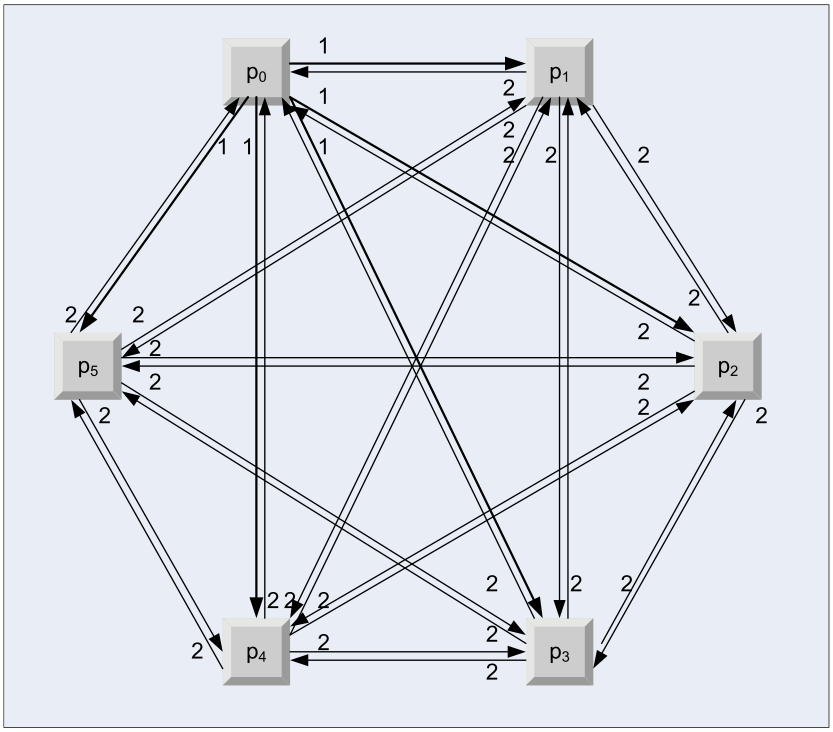

FIGURE 10.22

Six processes executing the snapshot protocol.

Each "take snapshot" message crosses each channel exactly once, and every process p_i has made its contribution to the global state; a process records its state the first time it receives a "take snapshot" message and then stops executing the underlying computation for some time. Thus, in a fully connected network with n processes, the protocol requires $n \times (n-1)$ messages, one on each channel.

For example, consider a set of six processes, each pair of processes being connected by two unidirectional channels, as shown in Fig. 10.22. Assume that all channels are empty, $\xi_{i,j} = 0$, $i \in \{0, 5\}$, $j \in \{0, 5\}$, at the time when process p_0 issues the "take snapshot" message. The actual flow of messages is

- In step 0, p_0 sends to itself the "take snapshot" message.
- In step 1, process p_0 sends five "take snapshot" messages labeled (1) in Fig. 10.22.
- In step 2, each of the five processes, p_1, p_2, p_3, p_4, and p_5 sends a "take snapshot" message labeled (2) to every other process.

A "take snapshot" message crosses each channel from process p_i to p_j, $i, j \in \{0, 5\}$ exactly once and $6 \times 5 = 30$ messages are exchanged.

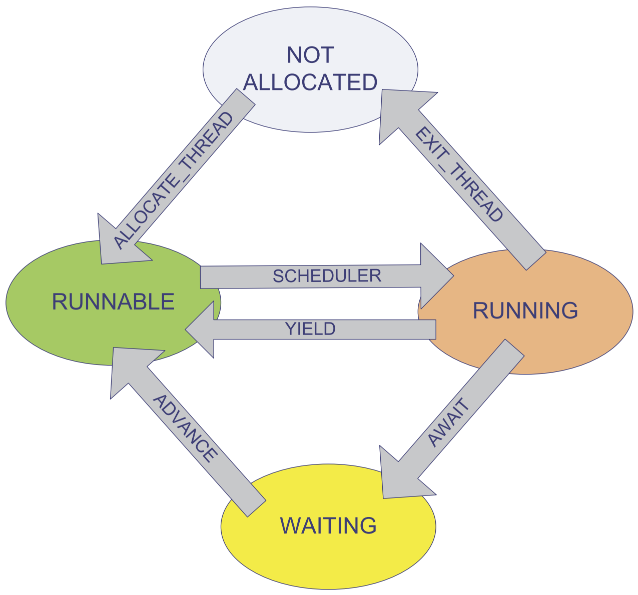

FIGURE 10.23

The state of a thread and the actions triggering a change of state.

10.11 Threads and activity coordination

While in the early days of computing, concurrency was analyzed mostly in the context of the system software, nowadays, concurrency is a ubiquitous feature of today's applications. Resources of a single server are insufficient for data-intensive applications and require a careful workload distribution to multiple instances running concurrently on a large number of servers; this is one of the main attractions of cloud computing. Introduction of multi-core processors was a disruptive event; the need to use effectively the cores of a modern processor forced many application developers to implement parallel algorithms and use multithreading.

Many concurrent applications are in the embedded systems area. Embedded systems are a class of reactive systems in which computations are triggered by external events. Such systems populate the Internet of Things (IoT) and are used by the critical infrastructure. A broad spectrum of such applications run multiple threads concurrently to control the ignition of cars, oil processing in a refinery, smart electric meters, heating and cooling systems in homes, or coffee makers. Embedded controllers for reactive real-time applications are implemented as mixed software–hardware systems.

Threads under the microscope. Threads are objects created explicitly to execute streams of instructions by an *allocate thread* action. A thread can be in several states, as shown in Fig. 10.23. Many threads share the core of a processor, and the system scheduler is the authority deciding when a thread gets control of the core and enters a *running* state.

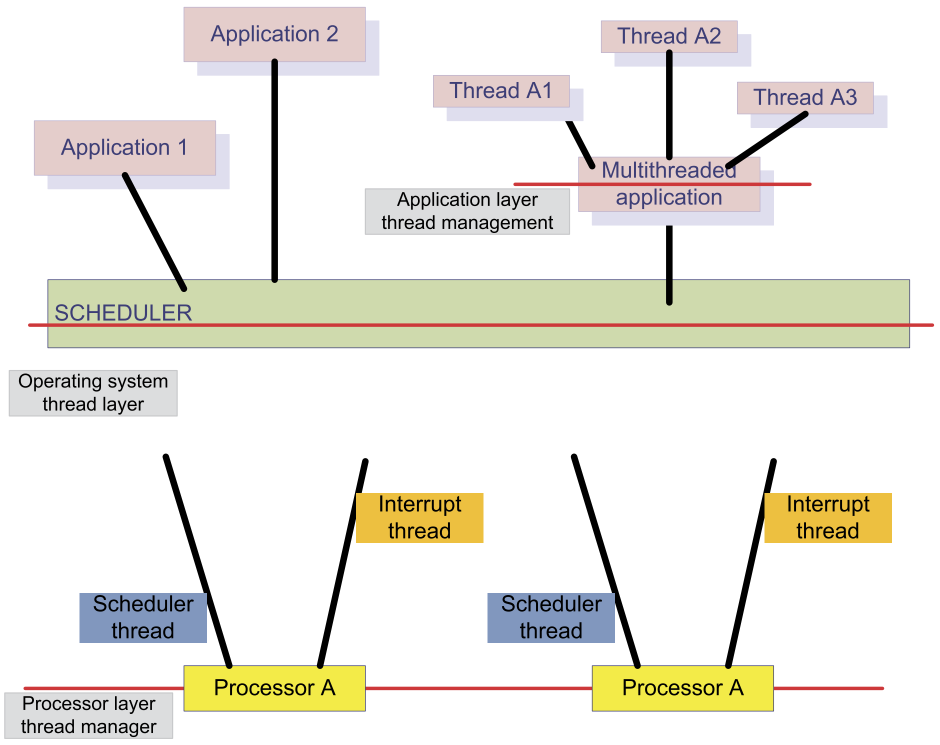

FIGURE 10.24

A snapshot of the thread population. Multiple application threads share a core under the control of a scheduler. Multiple operating system-level threads work behind the scene to carry out resource management functions.

The scheduler chooses from the pool of *runnable* threads, the only threads eligible to run. A running thread *yields* the control of the core when it has exhausted the time slot allocated to it or blocks by executing an *await* action, while waiting for the completion of an I/O operation and transition to a *wait* state. The thread could become *runnable* again when the scheduler decides to advance it, e.g., when the I/O operation has finished.

While one may be tempted to think only about application threads, the reality is that the kernel of the operating system operates also a fair number of threads to carry out its functions. Fig. 10.24 provides a snapshot of the thread population including application and operating system threads. Application-level multithreading enables the threads to share the resources allocated to the application, while operating system threads act behind the scene supporting a wide range of resource management functions.

The operation of the scheduler, while totally transparent to application developers, is fairly complex, and multithreading requires several context switches as seen in Fig. 10.25. A context switch of an application thread involves saving the thread state, including registers and the address used when the execution of the suspended thread becomes *runnable* again. Threads are lightweight entities, unlike

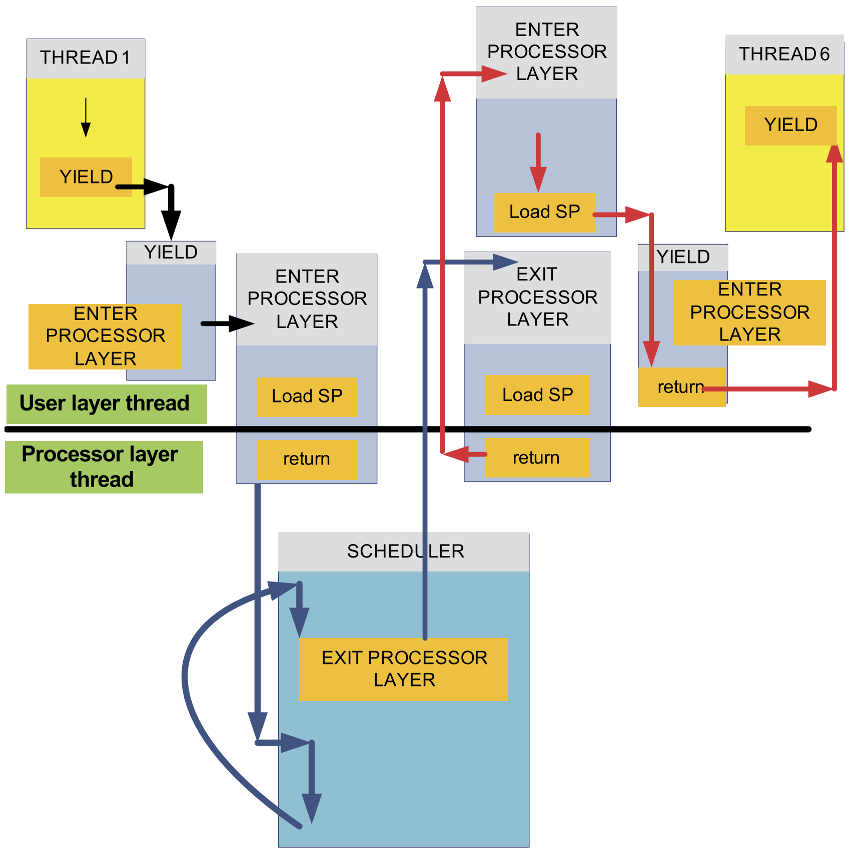

FIGURE 10.25

Application thread scheduling involves multiple context switches. A context switch saves the current thread state on the system stack. SP is the *stack pointer*.

processes in which information related to the address space, including pointers to the page tables and the process control block, are part of the process state and must be also saved.

Concurrency—the system software side. The kernel of an operating system exploits concurrency for virtualization of system resources such as the processor and the memory. *Virtualization*, covered in depth in Chapter 5, is a system design strategy with a broad range of objectives including:

- Hiding latency and performance enhancement, e.g., schedule a ready-to-run thread when the current thread is waiting for the completion of an I/O operation.
- Avoiding limitations imposed by physical resources, e.g., allow an application to run in a virtual address space of a standard size, rather than be restricted by the physical system memory size.
- Enhancing reliability and performance, as in the case of RAID systems mentioned in Section 2.6.

Sometimes, concurrency is used to describe activities that appear to be executed simultaneously, though only one of them may be active at any given time, as in the case of processor virtualization when multiple threads appear to run concurrently on a single processor. A thread can be suspended due to an external event, and a context switch to a different thread takes place. The state of the suspended thread is saved, the state of another thread ready to proceed is loaded, and then the thread is activated. The suspended thread will be reactivated at a later point in time.

Dealing with some of the effects of concurrency can be very challenging. Context switching could involve multiple components of a OS kernel, including the Virtual Memory Manager (VMM), the Exception Handler (EH), the Scheduler (S), and the Multi-level Memory Manager (MLMM). When a page fault occurs during the fetching of the next instruction, multiple context switches are necessary as shown in Fig. 10.26. The thread experiencing the fault is suspended, and scheduler dispatches another thread ready to run, while the exception handler invokes the multilevel memory manager.

If processor/core sharing seems complicated, operation of a multicore system running multiple virtual machines for applications running under different operating systems is even more complex. Now resource sharing occurs at the operating system level for the threads of one application running under that OS and at the hypervisor level for the threads of different virtual machines, as shown in Fig. 10.27.

Concurrency is often motivated by the desire to enhance the system performance. For example, in a pipelined computer architecture, multiple instructions are in different phases of execution at any given time. Once the pipeline is full, a result is produced at every pipeline cycle; an n-stage pipeline could potentially lead to a speedup by a factor of n. There is always a price to pay for increased performance, and in this example, the price is design complexity and cost. An n-stage pipeline requires n execution units, one for each stage, as well as a coordination unit. It also requires a careful timing analysis to achieve the full speedup.

This example shows that the management and the coordination of the concurrent activities increase the complexity of a system. The interaction between pipelining and virtual memory further complicates the functions of the kernel; indeed, one of the instructions in the pipeline could be interrupted due to a page fault and the handling of this case requires special precautions because the state of the processor is difficult to define.

Concurrency—the application software. Concurrency is exploited by application software to speedup a computation and to allow a number of clients to access a service. Parallel applications partition the workload and distribute it to multiple threads running concurrently. Distributed applications, including transaction management systems and applications based on the client-server paradigm discussed in Chapter 3, use extensively concurrency to improve the response time. For example, a web server spawns a new thread when a new request is received, thus multiple server threads run concurrently. A main attraction for hosting web-based applications is cloud elasticity, the ability of a service running on a cloud to acquire resources as needed and to pay for these resources as they are consumed.

Communication channels enable concurrent activities to work in concert and coordinate. Communication protocols enable us to transform noisy and unreliable channels into reliable ones that deliver messages in order. As mentioned earlier, concurrent activities communicate with one another via shared memory or via message passing. Multiple instances of a cloud application and a cloud service and its clients communicate via message passing. The Message Passing Interface (MPI) supports both synchronous and asynchronous communication, and it is often used by parallel and distributed applications.

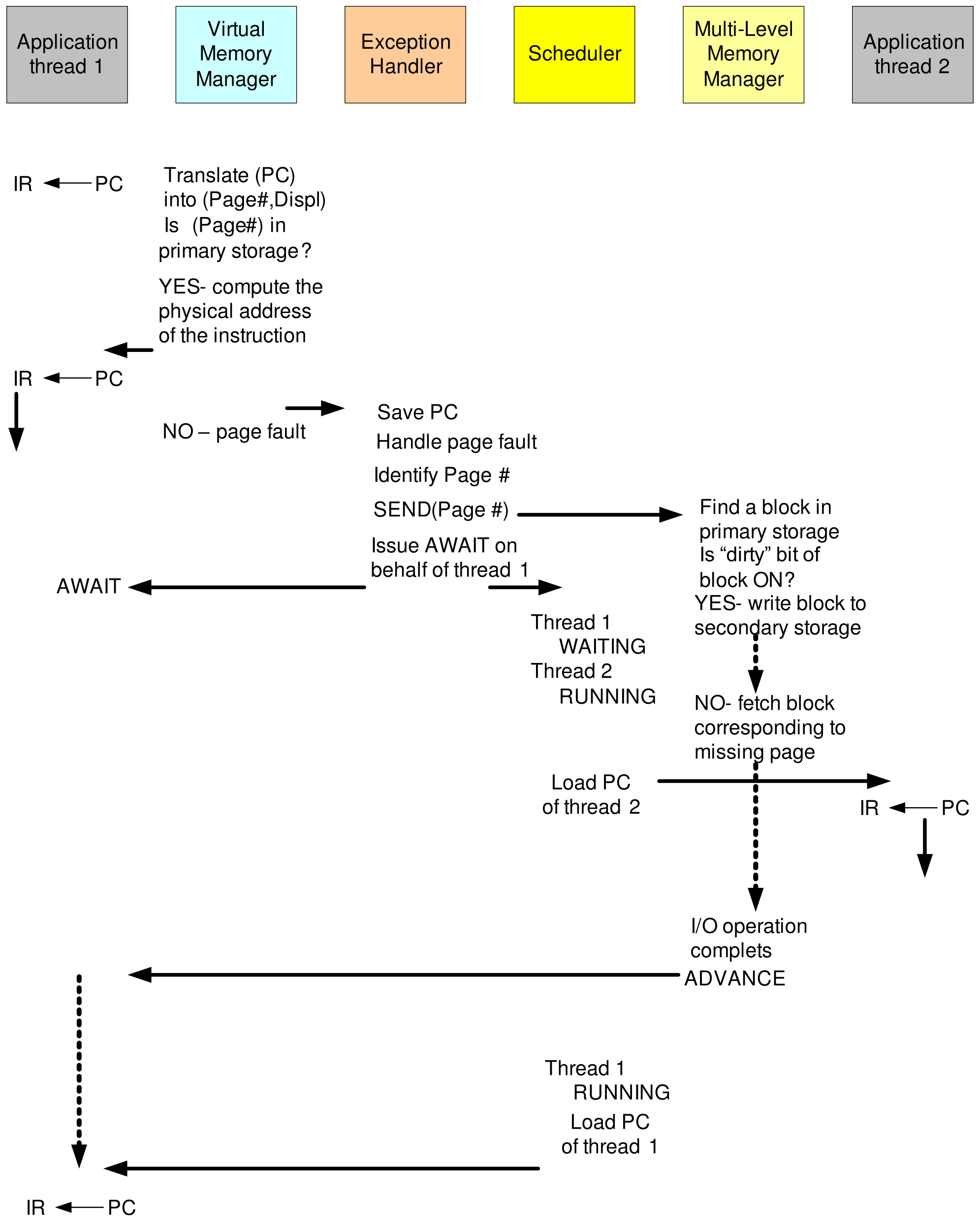
Application thread 1
Virtual Memory Manager
Exception Handler
Scheduler
Multi-Level Memory Manager
Application thread 2
IR ← PC
Translate (PC) into (Page#,Displ) Is (Page#) in primary storage?
YES- compute the physical address of the instruction
IR ← PC
NO – page fault
Save PC
Handle page fault
Identify Page #
SEND(Page #)
Find a block in primary storage Is "dirty" bit of block ON?
YES- write block to secondary storage
Issue AWAIT on behalf of thread 1
AWAIT
Thread 1 WAITING Thread 2 RUNNING
NO- fetch block corresponding to missing page
Load PC of thread 2
IR ← PC
I/O operation complets
ADVANCE
Thread 1 RUNNING
Load PC of thread 1
IR ← PC

FIGURE 10.26

Context switching when a page fault occurs during the instruction fetch phase. IR is the instruction register containing the current instruction, and PC is the program counter pointing to the next instruction to be executed. VMM attempts to translate the virtual address of the next instruction of thread 1 and encounters a page fault. Then, thread 1 is suspended waiting for an event when the page is brought in the physical memory from the disk. The Scheduler dispatches thread 2. To handle the fault, the Exception Handler invokes the MLMM.

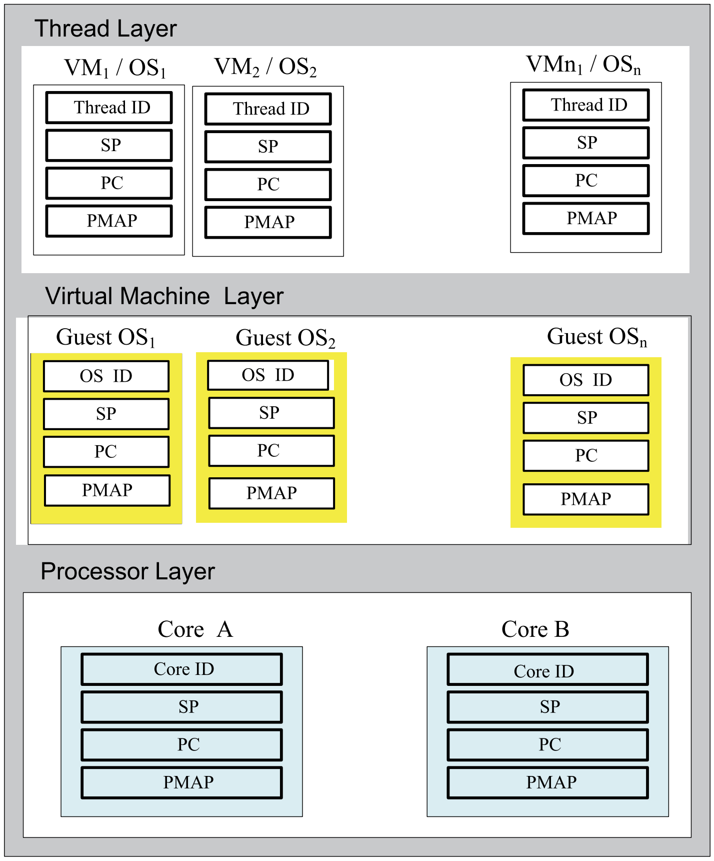

FIGURE 10.27

Thread multiplexing for a multicore server running multiple virtual machines. Multiple system data structures keep track of the contexts of all threads. The information required for context switching includes the ID, the stack pointer (SP), the program counter (PC), and the page table pointer (PMAP).

Message passing enforces modularity and prevents the communicating activities from *sharing their fate*; a server could fail without affecting the clients not using the service during the period the server was unavailable.

The communication patterns in the case of a parallel application are more structured, while patterns of communication for concurrent activities of a distributed application are more dynamic and unstructured. Barrier synchronization requires the threads running concurrently to wait until all of them have completed the current task before proceeding to the next. Sometimes, one of the activities, a coordinator, mediates communication among concurrent activities, while in other instances, individual threads communicate directly with one another.

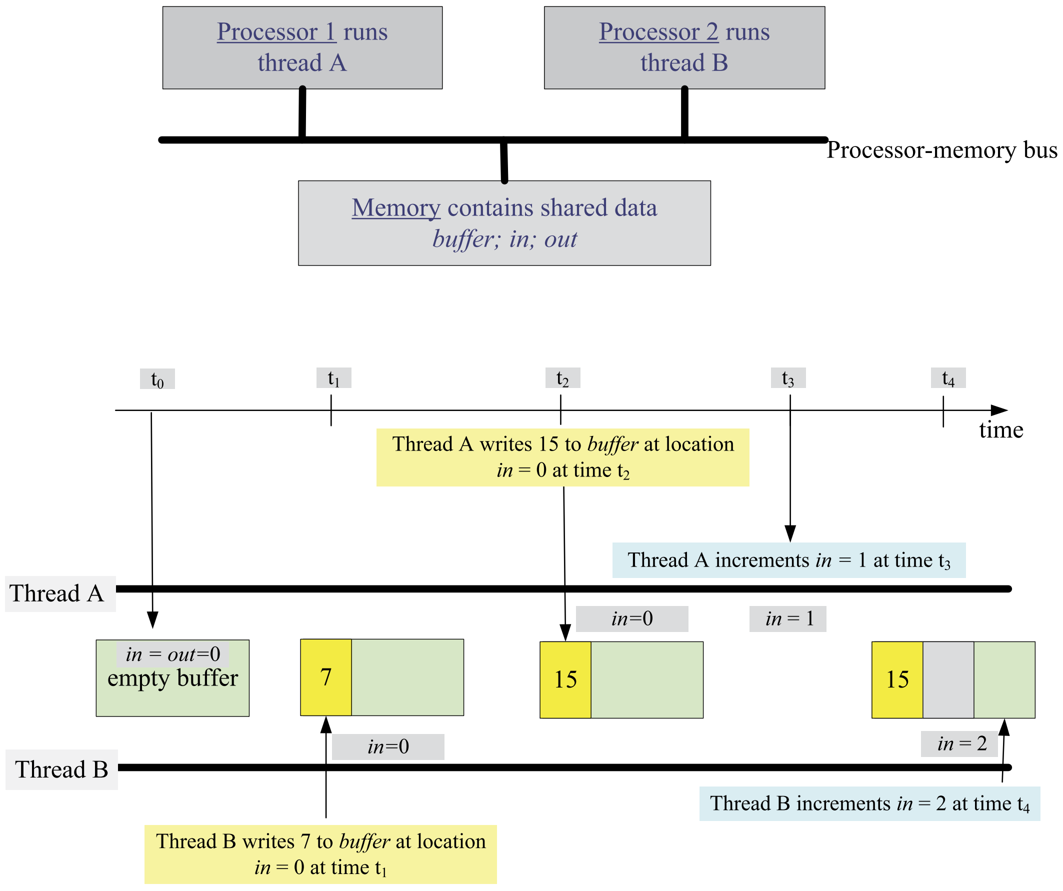
Processor 1 runs thread A
Processor 2 runs thread B
Processor-memory bus
Memory contains shared data buffer; in; out
t0
t1
t2
t3
t4
time
Thread A writes 15 to buffer at location in = 0 at time t2
Thread A increments in = 1 at time t3
Thread A
in=0
in = 1
in = out=0 empty buffer
7
15
15
in=0
in = 2
Thread B
Thread B increments in = 2 at time t4
Thread B writes 7 to buffer at location in = 0 at time t1

FIGURE 10.28

Race condition. Initially, at time t_0, the buffer is empty and $in = 0$. Thread B writes the integer 7 to the buffer at time t_1. Thread B is slow, incrementing the pointer *in* takes time and occurs at time t_4. In the meantime, at time $t_2 < t_4$, a faster thread A writes integer 15 to the buffer, overwrites the content of the first buffer location, and increments the pointer, $in = 1$ at time t_3. Finally, at time t_4, thread B increments the pointer $in = 2$.

Coordination of concurrent computations could be quite challenging and involves overhead that ultimately reduces the speedup of parallel computations. Concurrent execution could be very challenging, e.g., it could lead to *race conditions*, an undesirable effect in which the results of concurrent execution depend on the sequence of events. Fig. 10.28 illustrates a race condition in which two threads communicate using a shared data *buffer*. Both threads can write to the *buffer* location pointed at by *in* and can read from *buffer* location pointed at by *out*. When both threads attempt to write at about the same time, the item written by the second thread overwrites the item written by the first thread.

10.12 Critical sections, locks, deadlocks, and atomic actions

Parallel and distributed applications must take special precautions for handling shared resources. For example, consider a financial application when the shared resource is an account record. A thread running on behalf of a transaction first accesses the account to read the current balance, then updates the balance, and, finally, writes back the new balance.

If the thread is interrupted and another thread operating on the same account is allowed to proceed, before the first thread was able to complete the three steps for updating the account, the results of the financial transactions are incorrect. Another challenge is to deal with a transaction involving the transfer from one account to another. A system crash after operation completion on the first account will again lead to an inconsistency; the amount debited from the first account is not credited to the second.

In these cases, as in many other similar situations, a multistep operation should be allowed to proceed to completion without any interruptions, i.e., the operation should be *atomic*. An important observation is that such *atomic actions should not expose the state of the system until the action is completed.* Hiding the internal state of an atomic action reduces the number of states a system can be in, thus it simplifies the design and maintenance of the system. An atomic action is composed of several steps, each of which may fail. Therefore, we have to take additional precautions to avoid exposing the internal state of the system in case of a failure.

Locks and deadlocks. Concurrency requires a rigorous discipline when threads access shared resources. Concurrent reading of a shared data item is not restricted, while writing a shared data item should be subject to concurrency control. The race conditions discussed in Section 10.11 illustrate the problems created by a hazardous access to a shared resource, the buffer, that both threads attempt to write to. The hazards posed by a lack of concurrency control are ubiquitous. Imagine an embedded system at a power plant in which multiple events occur concurrently and the event signaling a dangerous malfunction of one subsystem is lost.

In all these cases, only one thread should be allowed to modify shared data at any given time, and other threads should only be allowed to read or write this data item only after the first one has finished. This process called *serialization* applies to segments of code called *critical sections* that need to be protected by control mechanisms called *locks,* permitting access to one and only one thread at a time.

A lock is an object that grants access to a critical section. To enter a critical section, a thread must acquire the lock of that section and after finishing must release the lock, as depicted in Fig. 10.29. Only one thread should be successful when multiple threads attempt to acquire the lock at the same time; the other threads must wait until the lock is released.

One may argue that serialization by locking a data structure is against the very nature of concurrency, allowing multiple computations to run at the same, but, without some form of concurrency control, it is not possible to guarantee the correctness of results of any computation. Lock-free programming [233] is rather challenging and will not be discussed in this chapter.

A lock should be seen as an antidote to uncontrolled concurrency and should be used sparingly and only to protect a critical section. Like any medication, locking has side effects; it does not only increase the execution time, but could lead to deadlocks. Indeed, another potential problem for concurrent execution of multiple processes/threads is the presence of deadlocks. A *deadlock* occurs when processes/threads competing with one another for resources are forced to wait for additional resources held by other processes/threads and none of the processes/threads can finish, as depicted in Fig. 10.30.

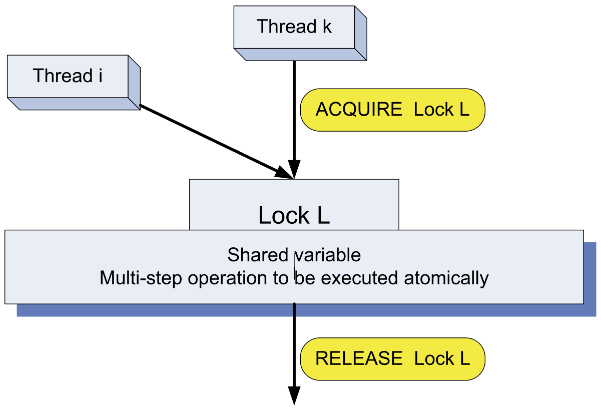

FIGURE 10.29

A lock protects a critical section consisting of multiple operations that have to be executed atomically.

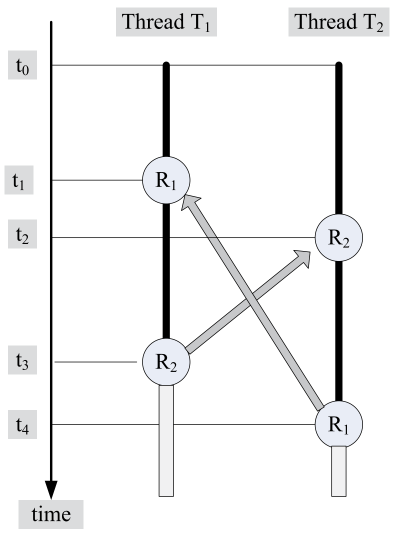

FIGURE 10.30

Thread deadlock. Threads T_1 and T_2 start concurrent execution at time t_0. Both need resources R_1 and R_2 to complete execution. T_1 acquires R_1 at time t_1 and T_2 acquires R_2 at time t_2. At time $t_3 > t_2$, thread T_1 attempts to acquire resource R_2 held by thread T_2 and blocks waiting for it to be released. At time $t_4 > t_3$, thread T_2 attempts to acquire resource R_1 held by thread T_1 and blocks waiting for it to be released. Neither thread can make any progress.

The four Coffman conditions [112] must hold simultaneously for a deadlock to occur:

1. *Mutual exclusion:* at least one resource must be nonsharable, and only one process or one thread may use the resource at any given time.

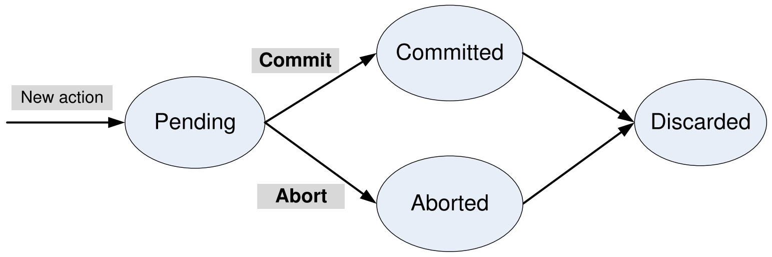

FIGURE 10.31

The states of an *all-or-nothing* action.

2. *Hold and wait:* at least one process or one thread must hold one or more resources and wait for others.
3. *Nonpreemption:* the scheduler or a monitor should not be able to force a process or a thread holding a resource to relinquish it.
4. *Circular wait;* given the set of n processes or threads $\{P_1, P_2, P_3, \ldots, P_n\}$, P_1 should wait for a resource held by P_2, P_2 should wait for a resource held by P_3, and so on, P_n should wait for a resource held by P_1.

There are other potential problems related to concurrency. When two or more processes/threads continually change their state in response to changes in the other processes, we have a *livelock* condition; the result is that none of the processes can complete its execution. Very often, processes/threads running concurrently are assigned priorities and scheduled based on these priorities. *Priority inversion* occurs when a higher priority process/task is indirectly preempted by a lower priority one.

Atomicity. The discussion of the transaction system suggests that an analysis of atomicity should pay special attention to the basic operation of updating the value of an object in storage. Even to modify the contents of a memory location, several machine instructions must be executed: load the current value in a register, modify the contents of the register, and store back the result.

Atomicity cannot be implemented without some hardware support; indeed, the instruction set of most processors support the *Test-and-Set* instruction that writes to a memory location and returns the old content of that memory cell as non-interruptible operations. Other architectures support *Compare-and-Swap*, an atomic instruction that compares the contents of a memory location to a given value and, only if the two values are the same, modifies the contents of that memory location to a given new value.

Two flavors of atomicity can be distinguished: *all-or-nothing* and *before-or-after* atomicity. *All-or-nothing* means that either the entire atomic action is carried out, or the system is left in the same state it was before the atomic action was attempted; in our examples a transaction is either carried out successfully, or the record targeted by the transaction is returned to its original state. The states of an *all-or-nothing* action are shown in Fig. 10.31.

To guarantee the all-or-nothing property of an action, we have to distinguish preparatory actions that can be undone, from irreversible ones, such as the alteration of the only copy of an object. Such preparatory actions are: allocation of a resource, fetching a page from secondary storage, allocation of memory on the stack, and so on. One of the golden rules of data management is never to change the

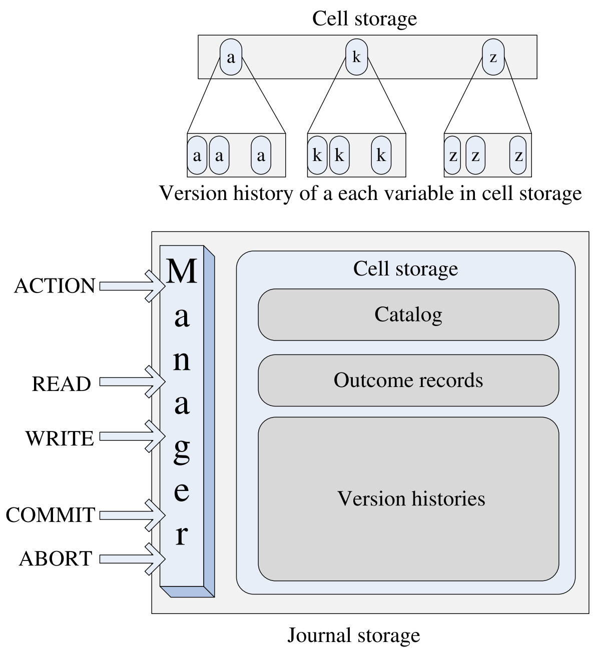

FIGURE 10.32

Storage models. Cell storage does not support all-or-nothing actions. When we maintain the version histories, it is possible to restore the original content, but we need to encapsulate the data access and provide mechanisms to implement the two phases of an atomic all-or-nothing action. The journal storage does precisely that.

only copy; maintaining the history of changes and a log of all activities allow us to deal with system failures and to ensure consistency.

An all-or-nothing action consists of a *pre-commit* and a *post-commit* phase; during the former it should be possible to backup from it without leaving any trace, while the latter phase should be able to run to completion. The transition from the first to the second phase is called a *commit point*. During the *pre-commit* phase, all steps necessary to prepare the post-commit phase, e.g., check permissions, swap in main memory all pages that may be needed, mount removable media, and allocate stack space, must be carried out; during this phase, no results should be exposed, and no actions that are irreversible should be carried out. Shared resources allocated during the pre-commit cannot be released until after the commit point. The commit step should be the last step of an all-or-nothing action.

A discussion of storage models illustrates the effort required to support all-or-nothing atomicity; see Fig. 10.32. The common storage model implemented by hardware is the so-called *cell storage*, a collection of cells each capable to hold an object, e.g., the primary memory of a computer in which each cell is addressable. Cell storage does not support all-or-nothing actions; once the contents of a cell is changed by an action, there is no way to abort the action and restore the original content of the cell.

To be able to restore a previous value, we have to maintain a *version history* for each variable in the cell storage. The storage model that supports all-or-nothing actions is called *journal storage*. Now, the cell storage is no longer accessible to the action, but the access is mitigated by a *storage manager*. In addition to the basic primitives to *Read* an existing value and to *Write* a new value in cell storage, the storage manager uniquely identifies an action that changes the value in cell storage and, when the action is aborted, is able to retrieve the version of the variable before the action and restore it. When the action is committed, then the new value should be written to the cell.

Fig. 10.32 shows that for a journal storage, in addition to the version histories of all variables affected by the action, we have to implement a catalog of variables and also to maintain a record to identify each new action. A new action first invokes the *Action* primitive; at that time, an outcome record uniquely identifying the action is created. Then, every time the action accesses a variable, the version history is modified and, finally, the action either invokes a *Commit* or an *Abort* primitive. In the journal storage model, the action is atomic and follows the state transition diagram in Fig. 10.31.

Before-or-after atomicity means that, from the point of view of an external observer, the effect of multiple actions is the same as if these actions have occurred one after another, in some order; a stronger condition is to impose a sequential order among transitions. In our example, the transaction acting on two accounts should either debit the first account and then credit the second one, or leave both accounts unchanged. The order is important because the first account cannot be left with a negative balance.

Atomicity is a critical concept for our efforts to build reliable systems from unreliable components and, at the same time, to support as much parallelism as possible for better performance. Atomicity allows us to deal with unforeseen events and to support coordination of concurrent activities. The unforeseen event could be a system crash, a request to share a control structure, the need to suspend an activity, and so on; in all these cases, we have to save the state of the process or of the entire system to be able to restart it at a later time.

As atomicity is required in many contexts, it is desirable to have a systematic approach rather than an ad hoc one. A systematic approach to atomicity must address several delicate questions:

a. How to guarantee that only one atomic action has access to a shared resource at any given time?
b. How to return to the original state of the system when an atomic action fails to complete?
c. How to ensure that the order of several atomic actions leads to consistent results?

Answers to these questions increase the complexity of the system and often generate additional problems. For example, access to shared resources can be protected by locks, but, when there are multiple shared resources protected by locks, concurrent activities may deadlock. A *lock* is a construct that enforces sequential access to a shared resource; such actions are packaged in the *critical sections* of the code. If the lock is not set, a thread first locks the access, then enters the critical section, and finally unlocks it; a thread wishing to enter the critical section finds the lock set and waits for the lock to be reset. A lock can be implemented using the hardware instructions supporting atomicity.

Semaphores and monitors are more elaborate structures ensuring serial access. Semaphores force processes to queue up when the lock is set and are released from this queue and allowed to enter the critical section one by one. Monitors provide special procedures to access the shared data; see Fig. 10.33. *The mechanisms for the process coordination we described require the cooperation of all activities,* the same way traffic lights prevent accidents only as long as the drivers follow the rules.

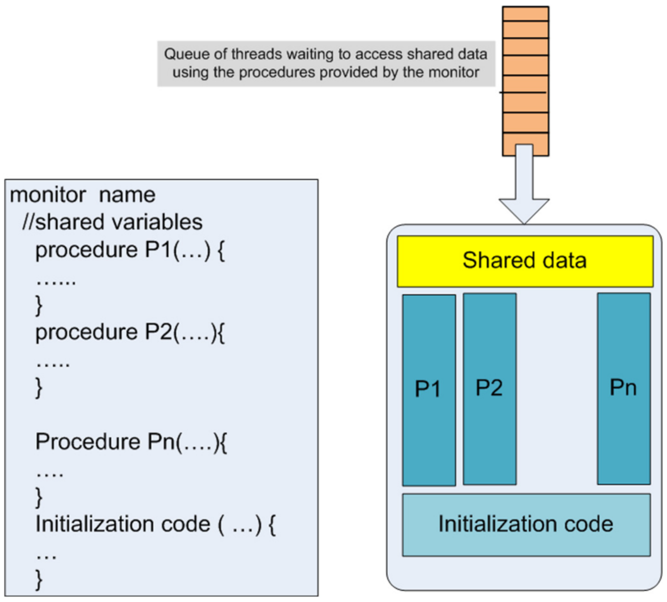

FIGURE 10.33

A monitor provides special procedures to access the data in a critical section.

10.13 Consensus protocols

Consensus is a pervasive problem in many areas of human endeavor; consensus is the process of agreeing to one of several alternates proposed by a number of agents. We restrict our discussion to a distributed system when a set of processes must reach consensus on a single proposed value.

No fault-tolerant consensus protocol can guarantee progress [175], but protocols that guarantee freedom from inconsistencies (safety) have been developed. A family of protocols to reach consensus based on a finite state machine approach is called *Paxos*.[5]

A fair number of contributions to the family of Paxos protocols are discussed in the literature. Leslie Lamport proposed several versions of the protocol including Disk Paxos, Cheap Paxos, Fast Paxos, Vertical Paxos, Stoppable Paxos, Byzantizing Paxos by Refinement, Generalized Consensus and Paxos, and Leaderless Byzantine Paxos. Lamport also published a paper on the fictional part-time parliament in Paxos [292] and a layman's dissection of the protocol [293].

The *consensus service* consists of a set of n processes. *Clients* send requests to processes and propose a value and wait for a response; the goal is to get the set of processes to reach consensus on a

[5] Paxos is a small Greek island in the Ionian Sea; a fictional consensus procedure is attributed to an ancient Paxos legislative body. The island had a part-time parliament because its inhabitants were more interested in other activities than in civic work; "the problem of governing with a part-time parliament bears a remarkable correspondence to the problem faced by today's fault-tolerant distributed systems, where legislators correspond to processes and leaving the Chamber corresponds to failing" according to Leslie Lamport [292] (for additional papers, see http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html).

single proposed value. A *quorum* is a subset of all acceptors. A proposal has a proposal number pn and contains a value v. Several types of requests such as *prepare* and *accept* flow through the system.

The *basic Paxos* considers several types of entities: (a) *client,* an agent that issues a request and waits for a response; (b) *proposer,* an agent with the mission to advocate a request from a client, convince the acceptors to agree on the value proposed by a client, and act as a coordinator to move the protocol forward in case of conflicts; (c) *acceptor*, an agent acting as the fault-tolerant "memory" of the protocol; (d) *learner,* an agent acting as the replication factor of the protocol and taking action once a request has been agreed upon; and finally, (e) the *leader,* a distinguished proposer.

In a typical deployment of the algorithm, an entity plays three roles, as proposer, acceptor, and learner. Then, the flow of messages can be described as follows [293]: "clients send messages to a leader; during normal operations the leader receives the client's command, assigns it a new command number i, and then begins the i-th instance of the consensus algorithm by sending messages to a set of acceptor processes." By merging the roles, the protocol "collapses" into an efficient client–master–replica style protocol.

The *basic Paxos* protocol is based on several assumptions about the processors and the network:

- Processes run on processors and communicate through a network; the processors and the network may experience failures, but not Byzantine failures. *A Byzantine failure is a fault presenting different symptoms to different observers.* In a distributed system, a Byzantine failure could be an *omission failure*, e.g., a crash failure, failure to receive a request or to send a response; it could also be a *commission failure*, e.g., process a request incorrectly, corrupt the local state, and/or send an incorrect or inconsistent response to a request.
- Processors: (i) operate at arbitrary speeds; (ii) have stable storage and may rejoin the protocol after a failure; and (iii) can send messages to any other processor.
- The network: (i) may lose, reorder, or duplicate messages; and (ii) messages are sent asynchronously and may take arbitrarily long time to reach the destination.

A proposal consists of a pair of numbers, a unique proposal number and a proposed value, (pn, v); multiple proposals may propose the same value v. A value is chosen if a simple majority of acceptors have accepted it. We need to guarantee that at most one value can be chosen; otherwise, there is no consensus. The two phases of the algorithm are:

<u>Phase I.</u>

1. *Proposal preparation:* a proposer (the leader) sends a proposal $(pn = k, v)$. The proposer chooses a proposal number $pn = k$ and sends a *prepare message* to a majority of acceptors requesting:
   - that a proposal with $pn < k$ should not be accepted;
   - the $pn < k$ of the highest number proposal already accepted by each acceptor.
2. *Proposal promise:* An acceptor must remember the highest proposal number it has ever accepted and the highest proposal number it has ever responded to. The acceptor can accept a proposal with $pn = k$ if and only if it has not responded to a prepare request with $pn > k$; if it has already replied to a prepare request for a proposal with $pn > k$, then it should not reply. Lost messages are treated as an acceptor that chooses not to respond.

Phase II.

1. *Accept request:* if the majority of acceptors respond, then the proposer chooses the value v of the proposal as follows:

   - the value v of the highest proposal number selected from all the responses;
   - an arbitrary value if no proposal was issued by any of the proposers.

   The proposer sends an *accept request* message to a quorum of acceptors including $(pn = k, v)$
2. *Accept:* If an acceptor receives an *accept message* for a proposal with the proposal number $pn = k$, it must accept it if and only if it has not already promised to consider proposals with a $pn > k$. If it accepts the proposal, it should register the value v and send an *accept* message to the proposer and to every learner; if it does not accept the proposal, it should ignore the request.

Several algorithm properties are important to show its correctness: (1) a proposal number is unique; (2) any two sets of acceptors have at least one acceptor in common; and (3) the value sent out in Phase 2 of the algorithm is the value of the highest numbered proposal of all the responses in Phase 1.

Fig. 10.34 illustrates the flow of messages for the consensus protocol. A detailed analysis of the message flows for various failure scenarios and of the properties of the protocol can be found in [293]. We only mention that the protocol defines three safety properties: (1) nontriviality—the only values that can be learned are proposed values; (2) consistency—at most one value can be learned; and (3) liveness—if a value v has been proposed, eventually, every learner will learn some value, provided that sufficient processors remain non-faulty. Fig. 10.35 shows the message exchange when there are three actors involved.

Chubby, a locking service based on the Paxos algorithm, is discussed in Section 7.7. A distributed coordination system discussed in Section 11.4, the Zookeeper, borrows several ideas from the Paxos algorithm:

i. A leader proposes values to the followers.
ii. Leaders wait for acknowledgments from a quorum of followers before considering a proposal committed (learned);
iii. Proposals include epoch numbers, which are similar to ballot numbers in Paxos.

10.14 Load balancing

A general formulation of the load balancing problem is to evenly distribute $\mathcal{N}$ objects to $\mathcal{P}$ places. Another formulation of the load balancing is in the context of placing m balls into $n < m$ bins, chosen independently and uniformly at random. The question in this case is to find the maximum number of balls in any bin. Load balancing can also be formulated in the context of hashing as the problem of placing m items sequentially in $n < m$ bins; the task is determining the longest time for finding an item.

The load balancing problem in a distributed system is formulated as follows: given a set $\mathcal{T}$ of tasks, distribute them to a set of $\mathcal{P}$ processors, which compute at the same rate, such that only one task can run at any given time on one processor; there is no preemption and each task runs to completion. Knowing the execution time of each task, the question is how to distribute them to minimize the completion time. Unfortunately, load balancing, as well as scheduling problems, are NP-complete [188].

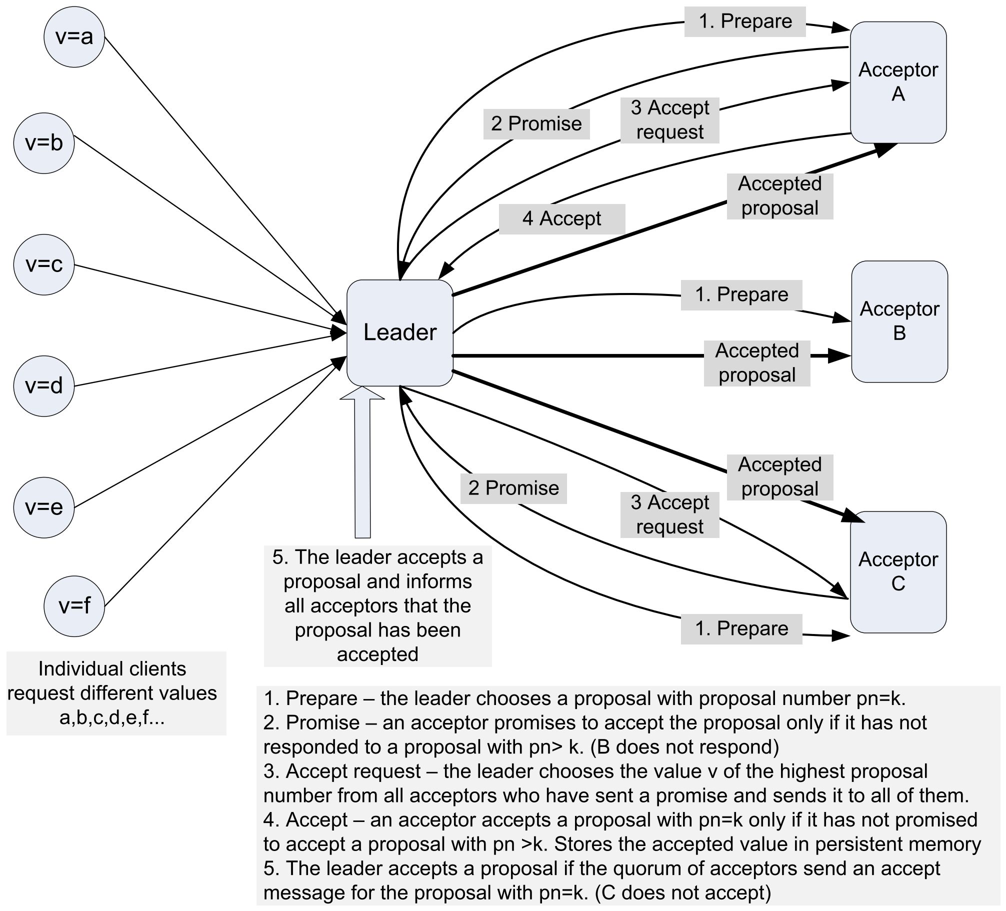
v=a
v=b
v=c
v=d
v=e
v=f
Individual clients request different values a,b,c,d,e,f...
Leader
1. Prepare
2 Promise
3 Accept request
4 Accept
Accepted proposal
Acceptor A
Acceptor B
Acceptor C
5. The leader accepts a proposal and informs all acceptors that the proposal has been accepted
1. Prepare – the leader chooses a proposal with proposal number pn=k.
2. Promise – an acceptor promises to accept the proposal only if it has not responded to a proposal with pn> k. (B does not respond)
3. Accept request – the leader chooses the value v of the highest proposal number from all acceptors who have sent a promise and sends it to all of them.
4. Accept – an acceptor accepts a proposal with pn=k only if it has not promised to accept a proposal with pn >k. Stores the accepted value in persistent memory
5. The leader accepts a proposal if the quorum of acceptors send an accept message for the proposal with pn=k. (C does not accept)

FIGURE 10.34

The flow of messages for the Paxos consensus algorithm. Individual clients propose different values to the leader who initiates the algorithm. Acceptor A accepts the value in message with proposal number pn = k; acceptor B does not respond with a promise, while acceptor C responds with a promise, but ultimately does not accept the proposal.

The importance of load balancing is undeniable, and practical solutions to overcome the algorithmic complexity are widely used. For example, randomization suggests to distribute the tasks to processors chosen independently and uniformly at random. This random distribution strategy should lead to an almost equal load of processors, provided that there are enough tasks and that the distribution of the task execution times is rather narrow. Several other heuristics are used in practice.

The balls-and-bins model. The load balancing problem is often discussed using the balls-and-bins model. In this model, we define the *load* of a bin as the number of balls in the bin. The question asked is: What is $\max(\mathcal{L}_i),\ 1 \le i \le n$, the maximum load in any bin, once all the n balls have chosen a

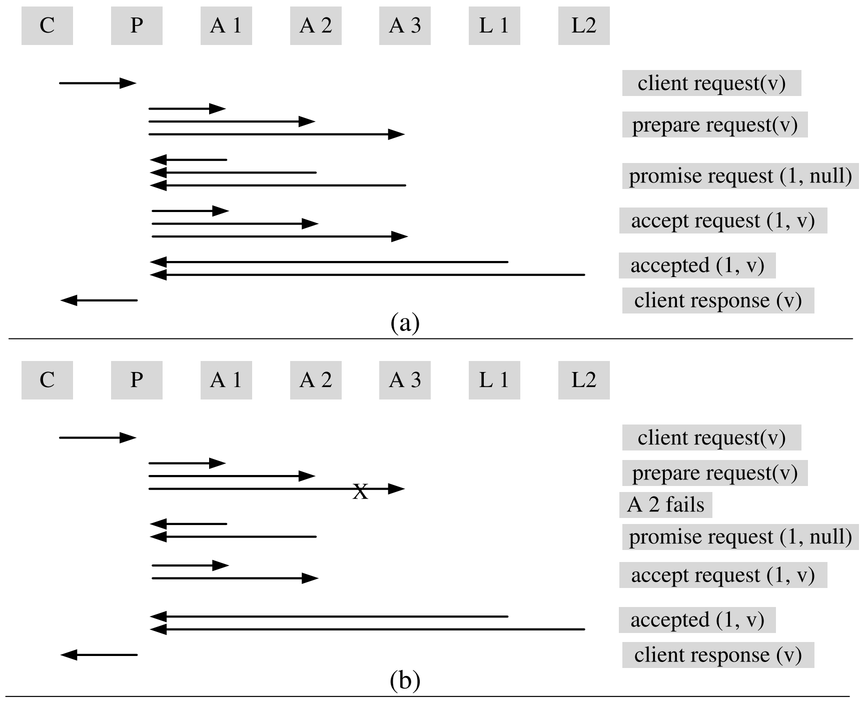

FIGURE 10.35

The basic Paxos with three actors: proposer (P), three acceptors (A1, A2, A3), and two learners (L1, L2). The client (C) sends a request to one of the actors playing the role of a proposer. The entities involved are: (a) successful first round when there are no failures; (b) successful first round of Paxos when an acceptor fails.

bin independently and uniformly at random? The answer is that with *high probability*, namely, with a probability $p \geq 1 - \mathcal{O}(1/n)$ [202],

$$\max(\mathcal{L}_i) \approx \frac{\log n}{\log \log n}. \tag{10.34}$$

This result applies also to task scheduling in a distributed system. Interestingly enough, this solution does not involve communication among the tasks, the processors, or among the tasks and the processors.

A rather surprising result proven in [39] is that better load balance is achieved when the balls are placed sequentially, and for each ball, we choose two bins independently and uniformly at random, then place the ball into the less full bin. In this case, the maximal load, $\max(\mathcal{L}_i^2)$, $1 \leq i \leq n$ is

$$\max(\mathcal{L}_i^2) \leq \frac{\log \log n}{\log 2} + \mathcal{O}(1) \ \text{ with high probability.} \tag{10.35}$$

This result, discussed in [353,355] and called *the power of two choices*, shows that having two or more choices leads to an exponential improvement of the load balance. The following discussion is based on the *layered induction approach* in [39], when the number of bins that contain at least j balls conditioned on the number of bins that contain at least $(j-1)$ balls is inductively bound.

If the choice for each ball is extended from 2 to d bins, then the result is further improved. The GREEDY algorithm in [39] considers an (m, n, d) problem: n initially empty bins, m balls to be placed sequentially in the bins, and d choices made independently and uniformly at random with replacement. Each ball is placed in the least loaded of the d bins, and ties are broken arbitrarily. Then, the maximum load, the maximum number of balls in a bin, has an upper bound of

$$\max(\mathcal{L}_i^d) \leq \frac{\log\log n}{\log d} + \mathcal{O}(1) \text{ with high probability.} \tag{10.36}$$

An intuitive justification of this results is discussed next. Call β_k the number of bins with *at least* k balls stacked on top of one another in the order they have been placed in the bin. The *height* of the top ball in a bin with k balls is k.

We wish to determine β_{k+1}, a high probability upper bound of the cardinality of bins loaded with at least $k+1$ balls. A bin will contain at least $k+1$ balls if in the previous round, it had at least k balls. Recall that there are at least β_k such bins; thus the probability of choosing a bin with k or more balls from the set of n bins is β_k/n. But there are $d > 2$ choices, therefore the probability that a ball lands in a bin already containing k or more balls drops at each step at least quadratically and is

$$p_{n,k,d} = \left(\frac{\beta_k}{n}\right)^d. \tag{10.37}$$

The number of balls with height at least $(k+1)$ is dominated by a Bernoulli random variable with the probability of success equals to $p_{n,k,d}$. This implies that

$$\beta_{k+1} \leq c \times \left[n \times \left(\frac{\beta_k}{n}\right)^d \right] \tag{10.38}$$

with c a constant. It follows that, after $j = \mathcal{O}(\log\log n)$ steps, the fraction β_k/n drops below $1/n$; thus, $\beta_j < 1$.

The sequential ball placement required in the algorithms discussed in this section and the decision to choose one of two bins, or one of d bins, deserves further scrutiny. It implies either a centralized system where an agent makes the choice, or some form of communication between balls to reach an agreement about the placement strategy. Communication is expensive, e.g., during the time of a short message exchange, a modern processor could execute several billion floating-point operations.

A tradeoff between load balance and communication is inevitable; we can reduce the maximum load only through coordination, thus the price to pay is increased communication. In a distributed system, a server is not aware of the tasks it has not communicated with, while tasks are unaware of the actions of other tasks and may only know the load of the servers. Global coordination among tasks is prohibitively expensive.

Parallelization of the randomized load balance. One of the questions examined in [353,355] is how to parallelize the randomized load balance. Once again, an extended balls-and-bins model is used and the goal is to minimize the maximum load and the communication complexity. Each one of the m balls begins by choosing d out of the n bins as prospective destination.

The choices are made independently and uniformly at random with replacement of balls, and the final decision of the destination bin requires r rounds of communication, with two stages per round. At each stage communication is done in parallel using short messages including an ID or an index. In the first stage each ball sends messages to all prospective bins, and in the second one each bin sends messages to all balls the bin has received a message from. During the final round, balls commit to a bin.

The strategies analyzed are symmetric, all balls and bins use the same algorithm, and all possible destinations are chosen independently and uniformly at random. An algorithm is *asynchronous* if an entity, a ball or a bin, does not have to wait for a round to complete; it only has to wait for messages addressed to it, rather than waiting for messages addressed to another entity. A round is *synchronous* if a barrier synchronization is required between some pairs of rounds.

A load lower bound for a broad class of algorithms like the one in [39] with r-rounds of communication derived in [355] is

$$\Omega\left(\sqrt[r]{\frac{\log n}{\log\log n}}\right), \tag{10.39}$$

with at least constant probability. Therefore, no algorithm can achieve a maximum load $\mathcal{O}(\log\log n)$ with high probability in a constant number of communication rounds.

A random graph $G(v, e)$ is used to represent the model and to derive this result. Each bin is a vertex v in this graph, and each ball is an undirected edge, e. When $d = 2$, the vertices of the two edges of a ball correspond to the two prospective bins. There are no self-loops in this graph where S denotes the set of edges. Multiple edges correspond to two balls that have chosen the same pair of bins. Selection of a bin by a ball transforms the undirected edge representing the ball into a directed edge oriented toward the vertex, or bin, the ball chooses as its destination. The goal is to minimize the maximum in-degree over the set of all graph vertices to avoid conflicts.

$\mathcal{N}(e)$, the *neighborhood* of an edge $e \in S$, is the set of all edges incident to an endpoint of e, and

$$\mathcal{N}(S) = \cup_{e\in S}\mathcal{N}(e). \tag{10.40}$$

Similarly, $\mathcal{N}(v)$ is the *neighborhood* of a vertex v. $\mathcal{N}_l(e)$, the *l-neighborhood* of an edge $e \in S$ is defined inductively as

$$\mathcal{N}_1(e) = \mathcal{N}(e), \quad \mathcal{N}_l(e) = \mathcal{N}(\mathcal{N}_{l-1}(e)). \tag{10.41}$$

$\mathcal{N}_{l,x}(e)$, the *(l, x)-neighborhood* of an edge $e(x, y)$, is defined inductively as

$$\mathcal{N}_{1,x}(e) = \mathcal{N}(x) - \{e\} \;\; \mathcal{N}_{l,x}(e) = \mathcal{N}(\mathcal{N}_{l-1,x}(e)) - \{e\}. \tag{10.42}$$

In an r round protocol, the balls make their choice in the final round; therefore each ball knows everything only about the balls in its $(r-1)$ neighborhood.

A ball $e = (x; y)$ learns from each bin about its *l-neighborhood* consisting of two subgraphs corresponding to $\mathcal{N}_{l,x}(e)$ and $\mathcal{N}_{l,y}(e)$. When the two subgraphs of the ball's *l-neighborhood* are isomorphic rooted trees, with the roots x and y, we say that the ball has a symmetric *l-neighborhood* or, that the ball is *confused*, and then the ball chooses the destination bin using a fair coin flip.

A tree of depth r, in which the root has degree T and each internal node has $T-1$ children, is called a (T, r)-*rooted, balanced tree*. A (T, r) tree in graph G is *isolated* if it is a connected component of G with no edges of multiplicity greater than one. A random graph with n vertices and n edges contains an isolated $(T, 2)$ tree with

$$T = \left(\sqrt{2} - \mathcal{O}(1)\right)\sqrt{\frac{\log n}{\log\log n}}, \tag{10.43}$$

with constant probability as shown in [355]. A corollary of this statement is that any nonadaptive, symmetric load distribution strategy for the balls-and-bins problem, with n balls and n bins and with $d=2$ and $r=2$, has a final load of at least

$$\left(\frac{\sqrt{2}}{2} - \mathcal{O}(1)\right)\sqrt{\frac{\log n}{\log\log n}}, \tag{10.44}$$

with at least constant probability. Indeed, half of the confused balls (edges) in an isolated $(T, 2)$ tree adjacent to the root will orient themselves towards the root because we have assumed that the balls flip an unbiased coin to choose the bin. This result can be extended for a range of r and d.

The balls-and-bins model has applications to hashing. The hashing implementation discussed in [277] uses a single hash function to map keys to entries in a table, and in case of a collision, i.e., when two or more keys map to the same table entry, all the colliding keys are stored in a linked list called a *chain*. The table entries are heads of chains and the longest search time occurs for the longest chain. The length of the longest chain is $\mathcal{O}(\frac{\log n}{\log\log n})$ with a high probability, when n keys are inserted into a table with n entries and each key is mapped to an entry of the table independently and uniformly, a process known as *perfect random hashing*.

The search time can be substantially reduced by using two hash functions and placing an item in the shorter of the two chains [266]. To search for an element, we have to search through the chains linked to the two entries given by both hash functions. If the n keys are sequentially inserted into the table with n entries, the length of the longest chain, thus, the maximum time to find an item, is $\mathcal{O}(\log\log n)$ with high probability.

The two-choice paradigm can be applied effectively to routing virtual circuits in interconnection networks with low congestion. The paradigm is also used to optimize the emulation of shared-memory multiprocessor (SMM) systems on a distributed-memory multiprocessor systems (DMM). The emulation algorithm should minimize the time needed by the DMM to emulate one step of the SMM.

The layered induction approach is also used in dynamic scenarios, e.g., when a new ball is inserted in the system [354]. Another technique to analyze load balancing based on the balls-and-bins model is the *witness tree*. To compute a bound for the probability of a "heavily-loaded" system event, we have to identify a *witness tree* of events and then estimate the probability that the witness tree occurs. This probability can be bounded by enumerating all possible witness trees and summing their individual probabilities of occurrence.

10.15 Multithreading in Java; FlumeJava; Apache Crunch

Java is a general-purpose computer programming language designed with portability in mind at Sun Microsystems.[6] Java applications are typically compiled to bytecode and can run on a Java Virtual Machine (JVM), regardless of the computer architecture. Java is a class-based, object-oriented language with support for concurrency. It is one of the most popular programming language, and it is widely used for a wide range of applications running on mobile devices and computer clouds.

Java Threads. Java supports processes and threads. Recall that a process has a self-contained execution environment and has its own private address space and run-time resources. A thread is a lightweight entity within a process. A Java application starts with one thread, the *main thread*, which can create additional threads.

Memory consistency errors occur when different threads have inconsistent views of the same data. Synchronized methods and synchronized statements are the two idioms for synchronization. Serialization of *critical sections* is protected by specifying the *synchronized* attribute in the definition of a class or method. This guarantees that only one thread can execute the critical section and each thread entering the section sees the modification done. Synchronized statements must specify the object that provides the intrinsic lock.

The current versions of Java support atomic operations of several datatypes with methods, such as *getAndDecrement(), getAndIncrement()* and *getAndSet()*. An effective way to control data sharing among threads is to share only immutable data among threads. A class is made *immutable* by marking all its fields as *final* and declaring the class as *final*.

A *Thread* in the *java.lang.Thread* class executes an object of type *java.lang.Runnable*. The *java.util.concurrent* package provides better support for concurrency than the *Thread* class. This package reduces the overhead for thread creation and prevents too many threads overloading the CPU and depleting the available storage. A *thread pool* is a collection of *Runnable* objects and contains a queue of tasks waiting to get executed.

Threads can communicate with one another via *interrupts*. A thread sends an interrupt by invoking an *interrupt* on the *Thread* object to the thread to be interrupted. The thread to be interrupted is expected to support its own interruption. *Thread.sleep* causes the current thread to suspend execution for a specified period.

The executor framework works with Runnable objects which cannot return results to the caller. The alternative is to use *java.util.concurrent.Callable*. A *Callable* object returns an object of type *java.util.concurrent.Future*. The *Future* object can be used to check the status of a *Callable* object and to retrieve the result from it. Yet, the *Future* interface has limitations for asynchronous execution, and the *CompletableFuture* extends the functionality of the *Future* interface for asynchronous execution.

Nonblocking algorithms based on low-level atomic hardware primitives such as compare-and-swap (CAS) are supported by Java 5.0 and later versions. The fork-join framework introduced in Java 7 supports the distribution of work to several workers and then waiting for their completion. The *join* method allows one thread to wait for completion of another.

[6] The design of Java was initiated in 1991 by James Gosling, Mike Sheridan, and Patrick Naughton with a C/C++-style syntax. Five principles guided its design: (1) simple, object-oriented, and familiar; (2) architecture-neutral and portable; (3) robust and secure; (4) interpreted, threaded, and dynamic; and (5) high performance. Java 1.0 was released in 1995. Java 8 is the only version currently supported for free by Oracle, a company that acquired Sun Microsystems in 2010.

FlumeJava. A Java library used to develop, test, and run efficiently data parallel pipelines is described in [90]. FlumeJava is used to develop data parallel applications such as MapReduce, discussed in Section 11.5.

At the heart of the system is the concept of *parallel collection*, which abstracts the details of data representation. Data in a parallel collection can be an in-memory data structure, one or more files, BigTable, discussed in Section 7.11, or a MySql database. Data-parallel computations are implemented by composition of several operations for parallel collections.

In turn, parallel operations are implemented using *deferred evaluation*. The invocation of a parallel operation records the operation and its arguments in an internal graph structure representing the execution plan. Once completed, the execution plan is optimized.

The most important classes of the FlumeJava library are the $Pcollection < T >$ used to specify an immutable bag of elements of type T and the $PTable < K, V >$ representing an immutable multimap with keys of type K and values of type V. The internal state of a $PCollection$ object is either *deferred* or *materialized*, i.e., not yet computed or computed, respectively. The $PObject < T >$ class is a container for a single Java object of type T and can be either deferred or materialized.

$parallelDo()$ supports element-wise computation over an input $PCollection < T >$ to produce a new output $PCollection < S >$. This primitive takes as the main argument a $DoFn < T, S >$, a function-like object defining how to map each value in the input into zero or more values in the output. In the following example from [90], $collectionOf(strings())$ specifies that the $parallelDo()$ operation should produce an unordered $PCollection$ whose $String$ elements should be encoded using UTF-8.[7]

```
Pcollection<String> words =
     lines.parallelDo(new DoFn<String, String> ( ) {
         void  process (String line, EmitFn<String> emitFn {
              for (String word : splitIntoWords(line) ) {
                    emitFn.emit(word);
               }
           }
       }, collectionOf(strings( ) ));
```

Other primitive operations are: $groupByKey()$, $combineValues()$ and $flatten()$.

- $groupByKey()$ converts a multimap of type $PTable < K, V >$. Multiple key/value pairs may share the same key into a unimap of type $PTable < K, Collection < V >>$, where each key maps to an unordered, plain Java Collection of all the values with that key.
- $combineValues()$ takes an input $PTable < K, Collection < V >>$ and an associative combining function on Vs, and returns a $PTable < K, V >$ where each input collection of values has been combined into a single output value.
- $flatten()$ takes a list of $PCollection < T >$s and returns a single $PCollection < T >$ that contains all the elements of the input $PCollections$.

[7] UTF-8 is a character encoding standard defined by Unicode capable of encoding all possible characters. The encoding is variable-length and uses 8-bit code units.

Pipelined operations are implemented by concatenation of functions. For example, if the output of function f is applied as input of function g in a $ParallelDo$ operation, then two $ParallelDo$ compute f and $f \otimes g$. The optimizer is only concerned with the structure of the execution plan and not with the optimization of user-defined functions.

FlumeJava traverses the operations in the plan of a batch application in forward topological order and executes each operation in turn. Independent operations are executed simultaneously. FlumeJava exploits not only the task parallelism but also the data parallelism within operations.

Apache Crunch, modeled after FlumeJava, simplifies creation of data pipelines on top of Apache Hadoop. Crunch was designed to use MapReduce effectively and is often used in conjunction with Hive and Pig, discussed in Section 11.8.

A Crunch pipeline creates a collection of Pig scripts and Hive queries. Crunch is fast, but slightly slower than a hand-tuned pipeline developed with the MapReduce APIs, according to https://attic.apache.org/projects/crunch.html.

10.16 History notes and further readings

In 1965, Edsger Dijkstra posed the problem of synchronizing N processes, each with a *critical section*, subject to two conditions: *mutual exclusion*—no two critical sections are executed concurrently; and *livelock freedom*—if several processes wait to enter critical sections, one of them will eventually succeed, [146]. Lamport comments "Dijkstra was aware from the beginning of how subtle concurrent algorithms are and how easy it is to get them wrong. He wrote a careful proof of his algorithm. The computational model implicit in his reasoning is that an execution is represented as a sequence of states, where a state consists of an assignment of values to the algorithm's variables plus other necessary information such as the control state of each process (what code it will execute next)."

Producer–consumer synchronization was the second fundamental concurrent programming problem identified by Dijkstra. An equivalent formulation of the problem is: Given a bounded FIFO (first-in-first-out), the producer stores data into an N-element buffer and the consumer retrieves the data. The algorithm uses three variables: N—the buffer size, in—the infinite sequence of unread input values, and out—the sequence of values output so far. In his discussion of the producer–consumer synchronization algorithm, Lamport notes that "The most important class of properties one proves about an algorithm are invariance properties. A state predicate is an invariant if it is true in every state of every execution."

Lamport notes that "Petri nets are a model of concurrent computation especially well-suited for expressing the need for arbitration. Although simple and elegant, Petri nets are not expressive enough to formally describe most interesting concurrent algorithms." He also mentioned that the first scientific examination of fault tolerance was Dijkstra's 1974 seminal paper on self-stabilization [147], a work ahead of its time. Arguably, the most influential study of concurrency models was Milner's Calculus of Communicating Systems (CCS) [348,349]. A number of formalisms based on the standard model were introduced for describing and reasoning about concurrent algorithms, including Amir Pnueli's temporal logic introduced in 1977 [400].

Further readings. A fair number of textbooks discuss theoretical, as well as practical, aspects of concurrency. For example, [509] is dedicated to concurrency in transactional processing systems, and [397] analyzes concurrency and consistency. The text [298] covers concurrent programming in Java, while [513] presents multithreading in C++.

The von Neumann architecture was introduced in [79]. The BSP and Multi-BSP models were introduced by Valiant in [485] and [486], respectively. Models of computations are discussed in [439]. PNs were introduced by Carl Adam Petri in [398]. An in-depth discussion of concurrency theory and system modeling with PNs can be found in [399]. The discussion of distributed systems leads to the observation that the analysis of communicating processes requires a more formal framework. Tony Hoare realized that a language based on execution traces is insufficient to abstract the behavior of communicating processes and developed *communicating sequential processes* (CSP) [241]. Milner initiated an axiomatic theory called the Calculus of Communicating System (CCS), [349]. Process algebra is the study of concurrent communicating processes within an algebraic framework. The process behavior is modeled as a set of equational axioms and a set of operators. This approach has its own limitations, the real-time behavior of the processes, the true concurrency, still escapes this axiomatization.

Seminal papers in distributed systems are authored by Many Chandy and Leslie Lamport [93], by Leslie Lamport [291], [292], [293], Tony Hoare [241], and Robin Milner [349]. The collection of contributions with the title "Distributed Systems," edited by Sape Mullender includes some of these papers. A survey of techniques and results related to the power of two random choices is presented in [354]. Seminal results on this subject are due to Azar [39], Karp [202,266], Mitzenmacher [353,355], and others.

10.17 Exercises and problems

Problem 1. Nonlinear algorithms do not obey the rules of scaled speed-up. For example, it was shown that, when the concurrency of an $\mathcal{O}(N^3)$ algorithm doubles, the problem size increases only by slightly more than 25%. Read [451] and explain this result.

Problem 2. Given a system of four concurrent threads t_1, t_2, t_3, and t_4 we take a snapshot of the consistent state of the system after 3, 2, 4, and 3 events in each thread, respectively; all but the second event in each thread are local events. The only communication event in thread t_1 is to send a message to t_4 and the only communication event in thread t_3 is to send a message to t_2. Draw a space–time diagram showing the consistent cut; mark individual events on thread t_i as e_i^j.
How many messages are exchanged to obtain the snapshot in this case? The snapshot protocol allows the application developers to create a checkpoint. An examination of the checkpoint data shows that an error has occurred, and it is decided to trace the execution. How many potential execution paths must be examined to debug the system?

Problem 3. The run-time of a data-intensive application could be days, or possibly weeks, even on a powerful supercomputer. Checkpoints are taken for a long-running computation periodically, and when a crash occurs, the computation is restarted from the latest checkpoint. This strategy is also useful for program and model debugging; when one observes wrong partial results, the computation can be restarted from a checkpoint where the partial results seem to be right. Express η, the *slowdown due to checkpointing*, for a computation in which checkpoints are taken after a run lasting τ units of time, and each checkpoint requires κ units of time. Discuss optimal choices for τ and κ. The checkpoint data can be stored locally, on the secondary storage of each processor, or on a dedicated storage server accessible via a high-speed network. Which solution is optimal and why?

Problem 4. What is in your opinion the critical step in the development of a systematic approach to all-or-nothing atomicity? What does a systematic approach mean? What are the advantages of a systematic versus an ad hoc approach to atomicity? The support for atomicity affects the complexity of a system. Explain how the support for atomicity requires new functions/mechanisms and how these new functions increase the system complexity. At the same time, atomicity could simplify the description of a system; discuss how it accomplishes this. Support for atomicity is critical for system features that lead to increased performance and functionality, such as: virtual memory, processor virtualization, system calls, and user-provided exception handlers. Analyze how atomicity is used in each case.

Problem 5. The PN in Fig. 10.10(d) models a group of n concurrent processes in a shared-memory environment. At any given time, only one process may write, but any subset of the n processes may read at the same time, provided that no process writes. Identify the firing sequences, the markings of the net, the postsets of all transition, and the presets of all places. Can you construct a state machine to model the same process?

Problem 6*. Consider a computation consisting of n stages with a barrier synchronization among the N threads at the end of each stage. Assuming that you know the distribution of the random execution time of each thread for each stage, show how one could use order statistics [128] to estimate the completion time of the computation.

Problem 7. Consider the data flow graph of a computation C in Fig. 10.6. Call t_1, t_2, t_3, and t_4 the time when the data inputs $data1$, $data2$, $data3$, and $data4$ become available. Call $T_i,\ 1 \leq i \leq 13$ the time required by computations $C_i,\ 1 \leq i \leq 13$, respectively, to complete and express T the total time required by C to complete function of t_i and T_i.

Problem 8. Discuss the factors affecting parallel slackness including characteristics of the parallel computation, such as fine versus coarse grain, and the characteristics of the workload and of the computing substrate.

Problem 9*. In Section 10.14, we discussed the *power of two choices* for the balls and bins problem with n bins. Instead of placing each ball in one random bin, we choose two random bins for each ball, place it in the one that currently has fewest balls, and proceed in this manner sequentially for each ball. Prove Eq. (10.35).

Hint—the idea of the proof: Call B_i the number of bins with more than i balls at the end. We wish to find an upper bound, β_i for B_i. The probability that a ball is placed in bin q with at least $i+1$ balls in it is $Pr(N_q \geq i+1) \leq \left(\frac{\beta_1}{n}\right)^2$. Indeed, both choices of placing this ball must be in bins with at least i balls. The distribution of bins B_{i+1} is dominated by the binomial distribution $Bin\left(n, \left(\frac{\beta_1}{n}\right)^2\right)$. The mean of this distribution is $\left(\frac{\beta_1}{n}\right)^2$. According to Chernoff bound $\beta_{i+1} = c\left(\frac{\beta_1}{n}\right)^2$ with some constant c. Therefore, $\frac{\beta_1}{n}$ decreases quadratically, and the following holds $i \approx \frac{\ln\ln n}{\ln 2} \Rightarrow \beta_1 < 1$. It follows that the maximum number of balls in a bin is $\frac{\ln\ln n}{\ln 2}$ with high probability.

Problem 10. What is the difference between wait for graph and resource allocation graph?