Chapter on Distributed Computing

Leslie Lamport and Nancy Lynch

February 3, 1989

Contents

Abstract

Rigorous analysis starts with a precise model of a distributed system; the most popular models, differing in how they represent interprocess communication, are message passing, shared variables, and synchronous communication. The properties satisfied by an algorithm must be precisely stated and carefully proved; the most successful approach is based on assertional reasoning. Algorithms for solving particular problems in a distributed system can then be designed and analyzed. Typical of the problems that have been addressed are concurrently accessing shared data, achieving consensus, analyzing network topology, obtaining consistent global information, and controlling database transactions.

1 What is Distributed Computing?

In the term *distributed computing*, the word *distributed* means spread out across space. Thus, distributed computing is an activity performed on a spatially distributed system. Although one usually speaks of a distributed system, it is more accurate to speak of a distributed *view* of a system. A hardware designer views an ordinary sequential computer as a distributed system, since its components are spread across several circuit boards, while a Pascal programmer views the same computer as nondistributed. An important problem in distributed computing is to provide a user with a nondistributed view of a distributed system—for example, to implement a distributed file system that allows the client programmer to ignore the physical location of his data.

We use the term *model* to denote a view or abstract representation of a distributed system. We will describe and discuss models informally, although we do present formal methods that can be used to reason about them.

The models of computation generally considered to be distributed are *process models*, in which computational activity is represented as the concurrent execution of sequential processes. Other models, such as Petri nets [Thi85], are usually not studied under the title of distributed computing, even though they may be used to model spatially distributed systems. We therefore restrict our attention to process models.

Different process models are distinguished by the mechanism employed for interprocess communication. The process models that are most obviously distributed are ones in which processes communicate by *message passing*—a process sends a message by adding it to a message queue, and another process receives the message by removing it from the queue. These models vary in such details as the length of the message queues and how long a delay may occur between when a message is sent and when it can be received. There are two significant assumptions embodied in message-passing models:

- Message passing represents the dominant cost of executing an algorithm.

- A process can continue to operate correctly despite the failure of other processes.

The first assumption distinguishes the use of message passing in distributed computing from its use as a synchronization mechanism in nondistributed concurrent computing. The second assumption characterizes the important subfield of fault-tolerant computing. Some degree of fault tolerance is required of most real distributed systems, but one often studies distributed algorithms that are not fault tolerant, leaving other mechanisms (such as interrupting the algorithm) to cope with failures.

Other process models are considered to be distributed if their interprocess communication mechanisms can be implemented efficiently enough by message passing, where efficiency is measured by the message passing costs incurred in achieving a reasonable degree of fault-tolerance. Algorithms exist for implementing virtually any process model by a message passing model with any desired degree of fault tolerance. Whether an implementation is efficient enough, and what constitutes a reasonable degree of fault tolerance are matters of judgement, so there is no consensus on what models are distributed.

2 Models of Distributed Systems

2.1 Message-Passing Models

2.1.1 Taxonomy

A wide variety of message-passing models can be used to represent distributed systems. They can be classified by the assumptions they make about four separate concerns: network topology, synchrony, failure, and message buffering. Different models do not necessarily represent different systems; they may be different views of the same system. An algorithm for implementing (or simulating) one model with another provides a mechanism for implementing one view of a system with a lower-level view. The entire goal of system design is to implement a simple and powerful user-level view with the lower-level view provided by the hardware.

Network Topology The network topology describes which processes can send messages directly to which other processes. The topology is described by a *communication graph* whose nodes are the processes, and where an arc from process i to process j denotes that i can send messages directly to j. Most models assume an undirected graph, where an arc joining two processes means that each can send messages to the other. However, one can also consider directed graph models in which there can be an arc from i to j without one from j to i, so i can send messages to j but not vice versa. We use the term *link* to denote an arc in the communication graph;

a message sent directly from one process to another is said to be sent over the link joining the two processes.

In some models, each process is assumed to know the complete set of processes, and in others a process is assumed to have only partial knowledge—usually the identity of its immediate neighbors. The simplest models, embodying the strongest assumptions, are ones with a completely connected communication graph, where each nonfaulty process knows about and can send messages directly to every other nonfaulty process. Routing algorithms are used to implement such a model with a weaker one.

Synchrony In the following discussion, all synchrony conditions are assumed to apply only in the absence of failure. Failure assumptions are treated separately below.

A *completely asynchronous* model is one with no concept of real time. It is assumed that messages are eventually delivered and processes eventually respond, but no assumption is made about how long it may take.

Other models introduce the concept of time and assume known upper bounds on message transmission time and process response time. For simplicity, in our examples we will use the simplest form of this assumption, that a message generated in response to an event at any time t (such as the receipt of another message) arrives at its destination by time $t + \delta$, where δ is a known constant.

Processes need some form of real-time clock to take advantage of this assumption. The simplest type of clock is a *timer*, which measures elapsed time; the instantaneous values of different processes' timers are independent of one another. Timers are used to detect failure, the assumption made above implying that a failure must have occurred if the reply to a message is not received within 2δ seconds of the sending of that message.

Some models make the stronger assumption that processes have synchronized clocks that run at approximately the correct rate of one second of clock time per second of real time. The simplest such assumption, which we use in our discussion, is that at each instant, the clocks of any two processes differ by at most ϵ for some known constant ϵ. Algorithms can use synchronized clocks to reduce the number of messages that need to be sent. For example, if a process is supposed to send a message at a known time t, then the receiving process knows that there must have been a failure if the message did not arrive by approximately time $t + \delta + \epsilon$ on its clock—the δ due to delivery time and the ϵ due to the difference between the two processes' clocks. Thus, one can test for failure by sending a single message rather than the query and response required with only timers. It appears to be a fundamental property of distributed systems that algorithms which depend upon synchronized clocks incur a delay proportional to the bound on clock differences (taken to be ϵ in our discussion).

Given a bound on the ratio of the running rates of any two processes' timers, and the assumed bound on message and processing delays, algorithms exist for constructing synchronized clocks from timers. These algorithms are discussed later.

The most strongly synchronous model is one in which the entire computation proceeds in a sequence of distinct rounds. At each round, every process sends messages, possibly to every other process, based upon the messages that it received in previous rounds. Thus, the processes act like processors in a single synchronous computer. This model is easily simulated using synchronized clocks by letting each round begin $\delta + \epsilon$ seconds after the preceding one.

Failure In message-passing models, one can consider both process failures and communication failures. It is commonly assumed that communication failure can result only in lost messages, although duplication of messages is sometimes allowed. Models in which incorrect messages may be delivered are seldom studied because it is believed that in practice, the use of redundant information (checksums) allows the system to detect garbled messages and discard them.

Models may allow transient errors that destroy individual messages, or they may consider only failures of individual links. A link failure may cause all messages sent over the link to be lost or, in a model with timers or clocks, a failed link may deliver messages too late. Since algorithms that use timers or clocks usually discard late messages, there is little use in distinguishing between late and lost messages. Of particular concern in considering link failures is whether or not one considers the possibility of *network partition*, where the communication graph becomes disconnected, making it impossible for some pairs of nodes to communicate with each other.

The weakest assumption made about process failure is that failure of one process cannot affect communication over a link joining two other processes, but any other behavior by the failed process is possible. Such models are said to allow *Byzantine* failure.

More restrictive models permit only *omission* failures, in which a faulty process fails to send some messages. (Since late messages are usually discarded, failures that cause a process to send messages too late can be considered omission failures.)

The most restrictive models allow only *halting* failures, in which a failed process does nothing. In the subclass of *fail-stop* models, other processes know when a process has failed [SA86].

In addition to the actual failure mode, some models make assumptions about how a failed process may be restarted. Models that allow only halting failures often assume some form of *stable storage* that is not affected by a failure. A failed process is restarted with its stable storage in the same state as before the failure and with every other part of its state restored to some initial values.

Failure models are problematic because it is difficult to determine how accurately they describe the behavior of real systems. It seems to be a widely held view among implementers of distributed systems that message loss and link failure adequately represent intercomputer communication failures. Whether or not a particular model of process failure is suitable depends upon the degree of reliability one requires of the system. There is general agreement that halting failure represents the most common type of computer failure—the familiar "system crash". It seems to provide a suitable model when only modest reliability is required. Omission faults, caused by unusual demand slowing down a computer's response time, should probably be considered when greater reliability is required. When extremely high reliability is required—especially when failure of the entire system could be life threatening—it seems necessary to assume Byzantine failures.

As we describe later, algorithms that tolerate Byzantine failures are more costly than ones that tolerate only more restricted failures. Less costly algorithms can be achieved by strengthening Byzantine failure models to allow *digital signatures* [DH79]. It is assumed that given an arbitrary data item D, any nonfaulty process i can generate a digital signature $S(i, D)$ such that any other process can determine whether a particular value v equals $S(i, D)$ for a given D, but no other process can generate $S(i, D)$. Although digital signatures are a cryptographic concept, in practical fault-tolerant algorithms they are implemented with redundancy. It is believed that, by the careful use of redundancy, the assumption made about digital signatures can be achieved with high enough probability to allow the use of the model even when extremely high reliability is required.

Message Buffering In message-passing models, there is a delay between when a message is sent and when it is received. Such a delay implies that there is some form of message buffering. Models may assume either finite or infinite buffers. With finite buffers, any link may contain only a fixed maximum number of messages that have been sent over that link but not yet received. When the link's buffer is full, attempts to send an additional message over the link either fail and produce some error response to the sending process or else cause the sending process to wait until there is room in the buffer. With infinite buffers, there may be arbitrarily many unreceived messages in a link's buffer, and the sender can always send another message over the link. Although any real system has a finite capacity, this capacity may be large enough to make infinite buffering a reasonable abstraction.

If a link's buffer can hold more than one message, it is possible for messages to be received in a different order than they were sent. Models with *FIFO* (first-in-first-out) buffering assume that messages that are not lost are always received in the same order in which they were sent. Many algorithms for asynchronous systems work only under the assumption of FIFO buffering. In most algorithms for systems with timers or synchronized clocks, a process does not send a message to another process until it knows that the previous message to that process has either been delivered or lost, so FIFO buffering need not be assumed. At the lowest level, real distributed systems usually provide FIFO buffering. This need not be the case at higher levels, where messages may be routed to their destination along multiple possible paths. However, if it is not provided by the underlying communication mechanism, FIFO buffering can be implemented by numbering the messages.

2.1.2 Measuring Complexity

There are two basic complexity measures for distributed algorithms: time and message complexity. The time complexity of an algorithm measures the time needed both for message transmission and for computation within the processes. However, computations performed by individual processes are traditionally ignored, only message-passing time being counted. This is a reasonable approximation for current computer networks in which message delivery time is usually several milliseconds or more, while computer operations are measured in microseconds. However, a millisecond is only a thousand microseconds, and a practical algorithm should not perform millions of extra calculations to save a few messages. Moreover, the large difference between message delivery time and processing time should not be taken for granted. Although it takes much longer for electromagnetic signals to travel within a processor than between processors in a spatially distributed system, current processing speed is limited primarily by circuit delays rather than transmission speed. With current technology, the high cost of sending a message is an artifact of the way systems are designed, since electrical signals can travel a kilometer in a few microseconds.

The usual measure of message-passing time for an algorithm is the length of the longest chain of messages that occurs before the algorithm terminates, where each message in the chain except the first is generated by the receipt of the previous one. For completely asynchronous models, where no assumptions are made about message delivery times, this seems to be the only reasonable way to measure worst-case message-passing time; for synchronous models that operate in rounds, it is just the number of rounds. The measure can be refined to take account of more precise timing assumptions—for example, if transmission delays are different for different links. Of course, processing time should be included in the time complexity if it is significant.

The most common measure of message complexity is the total number of messages transmitted. If messages contain on the order of a few hundred bits or more, then the total number of bits sent might be a better measure of the cost than the number of messages. In many algorithms, a process broadcasts the same message to n other processes. Depending upon the implementation details of the system, such a broadcast might cost as much as sending n separate messages, or it might cost no more than sending a single message.

Tradeoffs between time and message complexity are often possible. The minimal-time algorithm is usually simple, with more complex algorithms saving messages, but taking longer to terminate. It is often possible to "improve" algorithms by reducing their message complexity at the expense of their time complexity. However, many distributed systems contain few enough processes that an algorithm with a message complexity proportional to the square of the number of processes is quite practical and is often better than a more complicated one that uses fewer messages but takes longer.

As with sequential algorithms, there is also the question of whether to measure worst-case or average behavior—for example, whether to measure the maximum number of messages that can be sent or the expected number (in the sense of probability theory). When high reliability is required, worst-case behavior is usually the appropriate measure. In other cases, the average cost may be more important. Average costs have been derived mainly for probabilistic algorithms, in which processes make random choices.

2.2 Other Models

Other models of concurrent systems are usually described in terms of language constructs for interprocess communication. This can lead to the confusion of underlying concepts (what one says) with language issues (how one says it), but we know of no simple alternative for classifying the standard models.

2.2.1 Shared Variables

In the earliest models of concurrency, processes communicate through *global shared variables*—program variables that can be read and written by all processes. Initially, the shared variables were accessed by the ordinary program operations of expression evaluation and assignment; later variations included synchronization primitives such as semaphores [Dij68] and monitors [Hoa74] to control access to shared variables. Global shared variable models provide a natural representation of multiprocessing on a single computer with one or more processors connected to a central shared memory.

The most natural and efficient way to implement global shared variables with message passing is to have each shared variable maintained by a single process. That process can access the variable locally; it requires two messages for another process to read or write the variable. A read requires a query and a response with the value; a write requires sending the new value and receiving an acknowledgement that the operation was done—the acknowledgement is required because the correctness of shared-variable algorithms depends upon the assumption that a write is completed before the next operation is begun.

Such an implementation of global shared variables is not at all fault tolerant, since failure of the process holding the variable blocks the progress of any other process that accesses it. A fault-tolerant implementation must maintain multiple copies of the variable at different processes, which requires much more message passing. Hence, global shared variable models are not generally considered to be distributed.

A more restrictive class of models permits interprocess communication only through *local shared variables*, which are shared variables that are "owned" by individual processes. A local shared variable can be read by multiple processes, but it can be written only by the process that owns it. Reading a variable owned by a failed process is assumed to return some default value.

2.2.2 Synchronous Communication

Synchronous communication was introduced by Hoare in his Communicating Sequential Processes (CSP) language [Hoa78]. In CSP, process i sends a value v to process j by executing the *output command* $j!v$; process j receives that value, assigning it to variable x, by executing the *input command* $i?x$. Unlike the case of ordinary message passing, the input and output commands are executed synchronously. Execution of a $j!v$ operation is delayed until process i is ready to execute an $i?x$ operation, and vice versa. Thus, a CSP communication operation waits until a corresponding communication operation can be executed in another process.

There is an obvious way to implement synchronous communication with message passing. Process i begins execution of a $j!v$ command by sending a message to j with the value v; when process j is ready to execute the corresponding $i?x$ command, it sends an acknowledgement message to i and proceeds to its next operation. Process i can continue its execution when it receives the acknowledgement.

Many concurrent algorithms require that a process be prepared to communicate with any one of several processes, but actually communicate with only one of them before doing some further processing. With synchronous communication primitives, this means that a process must be prepared to execute any one of a set of input and/or output commands. If each process could be waiting for an arbitrary set of communication commands, then deciding which communications should occur could require a complicated distributed algorithm. For example, consider a network of three processes, each of which is ready to communicate with either one of the other two. Any pair of them can execute their corresponding communication actions, but only one pair may do so, and deciding upon that pair requires a distributed algorithm. To get around this difficulty, CSP allows a process to wait for an arbitrary set of input commands, but it may not be waiting for any other communication if it is ready to perform an output command. The choice of which communication to perform can then be made within a process, so each communication action requires only two messages.

Although CSP allows an efficient implementation with message passing, it does not permit fault tolerant algorithms. A process i that is waiting to execute a $j!v$ command cannot continue unless process j executes a corresponding $i?x$ command. The failure of process j therefore halts the execution of process i. (This could be avoided if i could wait to communicate with any one of several processes, which CSP prohibits.) Despite this difficulty, CSP is often considered a distributed model.

Closely related to synchronous communication is the *remote procedure call* or *rendezvous.* A remote procedure call is executed just like an ordinary procedure call, except the procedure is executed in another process. It can be implemented with two messages: one to send the arguments of the call in one message and another to return the result. Halting and omission failures can be handled by having the procedure call return an error result or raise an exception if no response to the first message is received. Remote procedure call is currently the most widely used language construct for implementing distributed systems without explicit message-passing operations.

2.3 Fundamental Concepts

The theory of sequential computing rests upon fundamental concepts of computability that are independent of any particular computational model. If there are any such fundamental formal concepts underlying distributed computing, they have yet to be developed. At present, the field seems to consist of a collection of largely unrelated results about individual models. Nevertheless, one can make some informal observations that seem to be important.

Underlying almost all models of concurrent systems is the assumption that an execution consists of a set of discrete events, each affecting only part of the system's state. Events are grouped into processes, each process being a more or less completely sequenced set of events sharing some common locality in terms of what part of the state they affect. For a collection of autonomous processes to act as a coherent system, the processes must be synchronized.

From the original work on concurrent process synchronization emerged two distinct classes of synchronization problem: contention and cooperation. The archetypical contention problem is the *mutual exclusion* problem, in which each process has a critical section and processes must be synchronized so that no two of them execute their critical section at the same time [Dij65]. As originally stated, this problem includes the requirement that a process be allowed to halt when not executing its critical section or its synchronization protocol. With this requirement, solutions are possible in shared-variable models but not in asynchronous message-passing models, which require that a process receive a message from every other process before it can enter its critical section. However, the mutual exclusion problem without this requirement has been studied in asynchronous message-passing systems.

The classic problem in cooperation is the *bounded buffer* problem, in which an unbounded sequence of values are transmitted in order from a sender process to a receiver process, using a fixed-length array of registers as a buffer. The receiver must wait when the buffer is empty, and the sender must wait when the buffer is full. This problem is best viewed as a symmetrical one, in which the sender generates filled buffer elements for use by the receiver and the receiver generates empty buffer elements for use by the sender.

The fundamental difference between these two forms of synchronization is that in contention problems a process must be able to make unlimited progress even if other processes fail to progress, while in cooperation problems the progress of one process depends upon the progress of another. For example, in the mutual exclusion problem, a process may enter its critical section an unlimited number of times while other processes are not requesting entrance, but in the bounded buffer problem, after the producer has filled the buffer it cannot proceed until the consumer creates an empty buffer element.

Problems of contention and cooperation appear in all models of concurrency. A class of problem that has arisen in the study of message-passing models is that of *global consistency*. For example, in a distributed banking system, one would like all branches of the bank to have a consistent view of the balance of any single account. In general, one would like to describe a distributed system in terms of its current global state. The global consistency problem is to ensure that all processes have a consistent view of the state. In the banking example, the amount of money currently in each account is part of the state.

To define a global state, there must be a total ordering of all transactions—to determine if there is enough money in my account for a withdrawal request to be granted, one must know if a deposit action occurred before or after the request. In an asynchronous message-passing model, there is no natural total ordering of events, only the partial ordering among events defined by letting event a precede event b if there is information flow permitting a to affect b. The definition of a global state requires completing the partial ordering of events, defined by the causality relation, to a total ordering. Achieving global consistency can be reduced to the problem of guaranteeing that all processes choose the same total ordering of events, thereby having the same definition of the global system state. One method of achieving this common total ordering is through the use of *logical clocks* [Lam78]. A logical clock is a counter maintained by each process with the property that if event

a precedes event b, then the time of event a precedes the time of event b, where the time of an event is measured on the logical clock of the process at which the event occurred. Logical clocks are implemented by attaching a *timestamp*, containing the current value of the sender's logical clock, to each message.

Because there is no unique definition of a global state in a message-passing model, it is sometimes mistakenly argued that one should not use the global state in reasoning about such models. The absence of a unique definition of the global state does not mean that we cannot reason in terms of an arbitrarily chosen definition. The method of reasoning we describe below, which involves reasoning about the state of a system, is useful for all concurrent models, including message-passing ones.

Another way of viewing the global consistency problem is in terms of knowledge. The problem exists because it is impossible for a process to know the current global state, since the concurrent activity of other processes can render its knowledge obsolete. It is rather natural to think about distributed algorithms in terms of what each process knows, and reasoning about the limitations on a process's knowledge forms the basis for proofs of many of the impossibility results described below. However, only recently has there been an attempt to perform this reasoning within formal theories of knowledge. [HM84]. These theories of knowledge provide a promising approach to a fundamental theory of distributed processing, but, at this writing, it is too early to know how successful they will prove to be.

3 Reasoning About Distributed Algorithms

Concurrent algorithms can be deceptive; an algorithm that looks simple may be quite complex, allowing unanticipated behavior. Rigorous reasoning is necessary to determine if an algorithm does what it is supposed to, and rigorous reasoning requires a formal foundation.

Here, we discuss *verification*—proving properties of concurrent algorithms. In verification, the properties to be proved are stated in terms of the algorithm itself—that is, in terms of the algorithm's variables and actions. The related field of *specification*, in which the properties to be satisfied are expressed in higher-level, implementation-independent terms, is considered briefly in Section 3.6. Specification methods must deal with the subtle question of what it means for a lower-level algorithm to implement a higher-level description. This question does not arise in the verification methods that we discuss, since the description of the algorithm and the properties to be proved are expressed in terms of the same objects.

3.1 A System as a Set of Behaviors

We have already seen that there are a wide variety of computational models of concurrent systems. However, they can almost all be described in terms of a single formal model, which forms the basis for our discussion of verification. In this model, we represent a concurrent system by a triple consisting of a set $\mathbf{S}$ of *states*, a set $\mathbf{A}$ of *actions*, and a set Σ of *behaviors*, each behavior being a finite or infinite sequence of the form

$$s_0 \stackrel{\alpha_1}{\rightarrow} s_1 \stackrel{\alpha_2}{\rightarrow} s_2 \ldots \tag{1}$$

where each s_i is a state and each α_i is an action. (If the sequence is finite, then it ends with a state s_n.) A state describes the complete instantaneous state of the system, an action is a system operation that is taken to be indivisible, and a behavior represents an execution of the system whose i^{th} action α_i takes the system from state s_{i-1} to state s_i. The set Σ represents the set of all possible system executions.

Most verification methods regard a behavior as either a sequence of states or a sequence of actions. Having states and actions in a behavior allows our discussion to apply to both approaches.

To reason about a system, one must first describe the triple $\mathbf{S}$, $\mathbf{A}$, Σ that represents it—for example, by a program in some programming language. Properties of the system are expressed by assertions about the set Σ. Here are three examples to indicate, very informally, how this is done.

mutual exclusion For every state s_i of every behavior of Σ, in s_i there is at most one process in its critical section. (For a state to be a complete description of the instantaneous state of the system, it must describe which processes are in their critical section.)

lockout-freedom: (This property asserts that a process that wants to enter its critical section eventually does so.) For every behavior of the form (1) in Σ and every $i \geq 0$, if s_i is a state in which a process is requesting entry to its critical section, then there is some $j > i$ such that s_j is a state in which that process is in its critical section.

bounded message delay: If α_i is the action of sending a message, s_{i-1} is a state in which the time is T, and s_j is a state in which the time is greater than $T + \delta$, then there is a k with $i < k < j$ such that α_k is the action of receiving that message. (This assumes that the current time is part of the state.)

3.2 Safety and Liveness

Any model is an abstraction that represents only some aspects of the system, and the choice of model restricts the class of properties one can reason about. Most formal reasoning about concurrent systems has been aimed at proving two kinds of properties: *safety* and *liveness*. Intuitively, a safety property asserts that something bad does not happen, and a liveness property asserts that something good eventually does happen.

In sequential computing, the most commonly studied safety property is partial correctness—if the program is started with correct input, then it does not terminate with the wrong answer, and the most commonly studied liveness property is termination—the program eventually terminates. A richer variety of safety and liveness properties are studied in concurrent computing; for example, mutual exclusion and bounded message delay are safety properties and lockout-freedom is a liveness property.

There are other classes of properties besides safety and liveness that are of interest—for example, the assertion that there is a .99 probability that the transmission delay is less than δ is neither a safety nor a liveness property. However, safety and liveness are the major classes of properties for which there are well developed methods of formal reasoning, so we will restrict our attention to them.

A safety or liveness property is an assertion about an individual behavior. It is satisfied by the system if it is true for all behaviors in Σ. A safety property is one that is false for a behavior if and only if it is false for some finite initial prefix of the behavior. (Intuitively, if something bad happens, then it happens after some finite number of actions.) A liveness property is one in which any finite behavior can be extended to a finite or infinite behavior (not necessarily a behavior of the program) that satisfies the property [AS85]. (Intuitively, after any finite portion of the behavior, it must still be possible for a good thing to happen.)

3.3 Describing a System

To give a formal description of a system, one must define the sets of states $\mathbf{S}$, actions $\mathbf{A}$, and behaviors Σ. A state is defined to be an assignment of values to some set of variables, where the variables may include ordinary program variables, message buffers, "program counters", and whatever else is needed to describe completely the instantaneous state of the computation. The set of actions is usually explicitly enumerated—for example, it may include all actions of the form *i sends m to j* for particular processes i and j and a particular message m. Actions represent internal operations of the system as well as input and output operations.

There are two general approaches to describing the set Σ. They may be called the *constructive* and *axiomatic* approaches, though we shall see that these names are misleading. In the constructive approach, one describes Σ by a program, where Σ is defined to be the set of all possible behaviors obtained by executing the program. The program may be written in a conventional programming language, or in terms of a formal model such as *I/O automata* [LT87] or Unity [CM88]. In the axiomatic approach, one describes Σ by a set of axioms, where Σ is defined to be the set of all sequences of the form of formula (1) that satisfy the axioms. The axioms may be written in a formal system—some form of temporal logic [Eme, Pnu77] being a currently popular choice—or in a less formal mathematical notation.

Axiomatic descriptions lead directly to a method of reasoning. If $\mathcal{S}$ is the set of axioms that describe Σ, and C is a property expressed in the same formal system as $\mathcal{S}$, then the system satisfies C if and only if the formula $\mathcal{S} \vdash \mathcal{C}$ is valid. On the other hand, constructive descriptions are often more convenient than axiomatic ones, since programming languages are designed especially for describing computations while formal systems are usually chosen for their logical properties.

In a constructive description, one specifies the possible state transitions $s \xrightarrow{\alpha} t$ caused by each action α of $\mathbf{A}$. A behavior of the form (1) is in Σ only if: (i) each transition $s_{i-1} \xrightarrow{\alpha_i} s_i$ is a possible state transition of α_i, and (ii) it is either infinite or it terminates in a state in which no further action is possible.

Formally, one defines a relation $\Gamma(\alpha)$ on $\mathbf{S}$ for each action α of $\mathbf{A}$, where $(s,t) \in \Gamma(\alpha)$ if and only if executing the action α in state s can produce state t. The action α is said to be *enabled* in state s if there exists some state t with $(s,t) \in \Gamma(\alpha)$. For example, the operation *send m to j* in process i's code is represented by the action α such that (s,t) is in $\Gamma(\alpha)$ if and only if s is a state in which control in process i is at operation α and t is the same as s except with m added to the queue of messages from i to j and with control in process i at the next operation after α; this action is enabled if and only if control is at the operation and the message queue is not full.

The behavior (1) is in Σ only if: (i) $(s_{i-1}, s_i) \in \Gamma(\alpha_i)$ for all i, and (ii) the sequence is either infinite or ends in a state s_n in which no action is enabled. Observe that condition (i) is a safety property.

In this definition, we include in Σ behaviors that start in any arbitrary state, including intermediate states one expects to encounter only in the middle of a computation and states that cannot occur in any computation. The properties one proves are of the form: if a behavior starts in a certain initial state, then .... For example, the set Σ for a mutual exclusion algorithm includes behaviors starting with several processes in their critical section. It is customary to include in the description a set of valid initial states, and to include in Σ only those behaviors starting in such a state. However, we find it more convenient not to assume any preferred starting states because, as we shall see, when proving liveness properties one must reason about the system's behavior starting from a point in the middle of the computation.

In addition to satisfying the two conditions above, sequences in Σ are usually required to satisfy some kind of fairness condition. For example, one may require that the sequence contain infinitely many actions from every process unless a point is reached after which no further actions of the process are enabled. This condition is expressed more formally by requiring that for every process k, either infinitely many of the α_i are actions of k or else there is some n such that no action of k is enabled in any state s_i with $i > n$. Fairness conditions are liveness properties.

In practice, fairness conditions do not affect the safety properties of a system. This means that if all behaviors in the set Σ described by a program satisfy a safety property C, then all behaviors satisfying only condition (i), with no fairness requirement, also satisfy C. Intuitively, safety properties are assertions about any arbitrarily long finite portion of the behavior, while liveness properties restrict only the infinite behavior. One can easily devise fairness conditions that affect safety properties—for example, the fairness requirement that every process executes infinitely many actions implies the safety property that no process ever reaches a halting state. However, such fairness conditions are unnatural and are never assumed in practice.

For reasoning about safety properties, one can therefore ignore condition (ii) and fairness conditions and consider only the relations $\Gamma(\alpha)$ defined by the actions. (In fact, (ii) really is a fairness condition.) Conversely, formal models that do not include fairness conditions are suitable only for studying safety properties, not liveness properties.

One can express conditions (i) and (ii) and the fairness conditions in a suitably chosen formal system. Expressing them in this way provides an axiomatic semantics for constructive descriptions, meaning that every program description in the form of a program can be translated into a collection of axioms. Thus, constructive descriptions can be viewed as a special class of axiomatic ones. In particular, we can adopt the simple approach to formal reasoning in which a program satisfies a property C if and only if $\mathcal{S} \vdash C$ is valid, where $\mathcal{S}$ is the translation of the program as a set of axioms. While this approach provides a formal definition of what it means for a program to satisfy a property, it does not necessarily provide a practical method for reasoning about programs because the axioms derived from conditions (i) and (ii) may be too complicated.

3.4 Assertional Reasoning

Verifying that a system satisfies a property C means showing that every behavior satisfying the definition of system behaviors also satisfies C. The obvious way of doing this is to reason directly about sequences, using either a temporal logic or direct mathematical reasoning about sequences. The problem with such an approach is that concurrent systems can exhibit a wide variety of possible behaviors. Reasoning directly about behaviors can become quite complex, with many different cases to consider. It is not clear if there are satisfactory methods for coping with this complexity.

Assertional methods attempt to overcome this difficulty by reducing the problem of reasoning about concurrent systems to that of reasoning separately about each individual action. In an assertional method, attention is concentrated on the states. A behavior is considered to be a (finite or infinite) sequence of states $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots$ and properties are expressed in terms of *state predicates*—boolean-valued functions on the set of states. Safety and liveness properties are handled by separate techniques.

3.4.1 Simple Safety Properties

It is convenient to introduce a bit of temporal logic to express properties. We interpret a state predicate P as an assertion about behaviors by defining P to be true for a behavior if and only if it is true for the first state of the behavior. We define $\Box P$ to be the assertion that is true for a behavior if and only if P is true for all states in the behavior, so $\Box P$ asserts that P is "always" true.

Traditional assertional methods prove safety properties of the form $P \Rightarrow$

$\Box Q$ for state predicates P and Q. Most safety properties that have been considered are of this form, with P being the predicate asserting that program control is at the beginning and all program variables have their correct initial values. For example, partial correctness is expressed by letting Q be the predicate asserting that if control is at the end then the variables have the correct final values, and mutual exclusion is expressed by letting Q be the predicate asserting that no two processes are in their critical sections. Proving such a property means showing that a certain class of states in $\mathbf{S}$, namely the states in which Q is false, do not appear in any behaviors in Σ that begin in a state with P true.

We say that a state predicate I is an *invariant* of a system if no action in $\mathbf{A}$ can make I false. More formally, I is an invariant if and only if for every action α in $\mathbf{A}$ and every pair (s, t) in $\Gamma(\alpha)$: if $I(s)$ is true then $I(t)$ is true. A simple induction argument shows that if I is an invariant then $I \Rightarrow \Box I$ is true for every behavior in Σ. (What we call an invariant is also called a *stable property*, and the term "invariant" is often used to mean a stable property that is true of the initial state.)

In assertional methods, one proves $P \Rightarrow \Box Q$ by finding a predicate I such that (i) I is an invariant, (ii) P implies I, and (iii) I implies Q. Since the invariance of I means that $I \Rightarrow \Box I$ is true for every sequence in Σ, it follows easily from (ii) and (iii) that $P \Rightarrow \Box Q$ is true for every sequence in Σ.

As a simple example, consider the two-process program in Figure 1, where each process cycles repeatedly through a loop composed of three statements, the angle brackets enclosing atomic actions. This program describes a common hardware synchronization protocol that ensures that the two processes alternately execute their critical sections. (For simplicity the critical sections are represented by atomic actions.) We prove that this algorithm guarantees mutual exclusion, which means that control is not at the *critical section* statements in both processes at the same time.[1] Mutual exclusion is expressed formally as the requirement $P \Rightarrow \Box Q$, where the predicates P and Q are defined by

$$\begin{aligned} P &\equiv at(\alpha) \wedge at(\lambda) \\ Q &\equiv \neg(at(\beta) \wedge at(\mu)) \ , \end{aligned}$$

$at(\alpha)$ is the predicate asserting that control in the first process is at state-

[1] This protocol does not solve the original mutual exclusion problem because one process cannot progress if the other halts.

variables x, y : **boolean**;

cobegin loop α: $\langle$ await $x = y\,\rangle$;
$\qquad\qquad\qquad\beta$: $\langle$ *critical section* $\rangle$;
$\qquad\qquad\qquad\gamma$: $\langle\, x := \neg x\,\rangle$
$\qquad\quad$ **end loop**

$\Box$

$\qquad\quad$ **loop** λ: $\langle$ await $y \neq x\,\rangle$;
$\qquad\qquad\qquad\mu$: $\langle$ *critical section* $\rangle$;
$\qquad\qquad\qquad\nu$: $\langle\, y := \neg y\,\rangle$
$\qquad\quad$ **end loop**

coend

Figure 1: A simple synchronization protocol.

ment α, and the other "*at*" predicates are similarly defined.

The invariant I used to prove this property is defined by

$$I \;\equiv\; ((at(\beta) \vee at(\gamma)) \Rightarrow (x = y)) \wedge ((at(\mu) \vee at(\nu)) \Rightarrow (x \neq y))$$

If the critical sections do not change x or y, then executing any atomic action of the program starting with I true leaves I true, so I is an invariant. It is also easy to check that $P \Rightarrow I$ and $I \Rightarrow Q$, which imply $P \Rightarrow \Box Q$.

The method of proving safety properties of the form $P \Rightarrow \Box Q$ can be generalized to prove properties of the form $P \wedge \Box R \Rightarrow \Box Q$ for predicates P, Q, and R. Such properties are used in proving liveness properties. We say that a predicate I is *invariant under the constraint* R if any action executed in a state with $I \wedge R$ true leaves I true or makes R false. If I is invariant under the constraint R, then $I \wedge \Box R \Rightarrow \Box I$ is true for every behavior in Σ. One can therefore prove $P \wedge \Box R \Rightarrow \Box Q$ by finding a predicate I such that (i) I is an invariant under the constraint R, (ii) P implies I, and (iii) I implies Q. Thus, the ordinary assertional method for proving $P \Rightarrow \Box Q$ is extended to prove properties of the form $P \wedge \Box R \Rightarrow \Box Q$ by replacing invariance with invariance under the constraint R.

The hard part of an assertional proof is constructing I and verifying that it is an invariant (or an invariant under a constraint). The predicate I can be quite complicated, and finding it can be difficult. However, proving that it is an invariant is reduced to reasoning separately about each individual action.

Experience has indicated that this reduction is usually simpler and more illuminating than reasoning directly about the behaviors for proving safety properties that are easily expressed in the form $P \Rightarrow \Box Q$. However, reasoning about behaviors has been more successful for proving properties that are not easily expressed in this form. It is usually the case that safety properties one proves about a particular algorithm are of the form $P \Rightarrow \Box Q$, while general properties one proves about classes of algorithms are not.

Because the invariant I can be complicated, one wants to decompose it and further decompose the proof of its invariance. This is done by the *Owicki-Gries method* [OG76], in which the invariant is written as a program annotation with predicates attached to program control points. In this method, I is the conjunction of predicates of the form "If program control is at this point, then the attached predicate is true." The decomposition of the invariance proof is based upon the following principle: if I is an invariant and I' is invariant under the constraint I then $I \wedge I'$ is an invariant.

A number of variations of the Owicki-Gries method have been proposed, usually for the purpose of handling particular styles of interprocess communication[AFdR80, LG81]. These methods are usually described in terms of proof rules—the individual steps one goes through in proving invariance—without explicitly mentioning I or the underlying concept of invariance. This has tended to obscure their simple common foundation.

3.4.2 Liveness Properties

If P and Q are predicates, then $P \leadsto Q$ is defined to be true if, whenever a state is reached in which P is true, then eventually a state will be reached in which Q is true. More precisely, $P \leadsto Q$ is true for the sequence (1) if for every n, if $P(s_n)$ is true then there exists an $m \geq n$ such that $Q(s_m)$ is true. Most liveness properties that one wishes to prove about systems are expressible in the form $P \leadsto Q$. For example, termination is expressed by letting P assert that the program is in its starting state and letting Q assert that the program has terminated; lockout-freedom is expressed by letting P assert that some process k is requesting entry to its critical section and letting Q assert that k is in its critical section.

The basic method of proving liveness properties is by a counting argument, using a *well-founded* set—one with a partial ordering relation $\succ$ such that there are no infinite chains of the form $e_i \succ e_2 \succ \ldots$. Suppose we construct a function w from the set of states to a well-founded set with the following property: if the system is in a state s in which $Q(s)$ is false, then it must eventually reach a state t in which either $Q(t)$ is true or $w(s) \succ w(t)$. Since the value of w cannot decrease forever, this implies that Q must eventually become true.

To prove $P \leadsto Q$, we construct such a function w and prove that it has the required property—namely, that its value must keep decreasing unless Q becomes true. In this proof, we may assume the truth of any predicate R such that $P \Rightarrow \Box R$ is true for all behaviors in Σ. This is a generalization of the usual method for proving termination of a loop in a sequential program, in which w decreases with each iteration of the loop and R asserts that the loop invariant[2] is true if control is at the start of the loop.

One still needs some way of proving that w must decrease unless Q becomes true, assuming the truth of a predicate R that satisfies $P \Rightarrow \Box R$. The simplest approach is to prove that each action in **A** either decreases the value of w or else makes Q true—in other words, that for every action α and every $(s,t) \in \Gamma(\alpha)$: $R(s) \wedge \neg Q(s)$ implies $w(s) \succ w(t) \vee Q(t) \vee \neg R(t)$.

This approach works only if the validity of the property $P \leadsto Q$ does not depend upon any fairness assumptions. To see how it can be generalized to handle fairness, consider the simple fairness assumption that if an action is continuously enabled, then it must eventually be executed—in other words, for every behavior (1) and every $n > 0$: if α is enabled in all states s_i with $i \geq n$, then $\alpha = \alpha_i$ for some $i > n$. Under this assumption, it suffices to show that every action either leaves the value of w unchanged or else decreases it, and that there is at least one action α whose execution decreases w, where α remains enabled until it is executed. Again, this need be proved only under the assumption that Q remains false and R remains true, where R is a predicate satisfying $P \Rightarrow \Box R$.

The problem with this approach is that the precise rules for reasoning depend upon the type of fairness assumptions. An alternative approach uses the single framework of temporal logic to reason about any kind of fairness conditions. We have already written the liveness property to be proved ($P \leadsto Q$) and the safety properties used in its proof (properties of the form $P \Rightarrow \Box R$) as temporal logic formulas. The fairness conditions are also expressible as a collection of temporal logic formulas. Logically, all that must be done is to prove, using the rules of temporal logic, that the fairness

[2]A loop invariant is not an invariant according to our definition, since it asserts only what must be true when control is at a certain point, saying nothing about what must be true at the preceding control point.

conditions and the safety properties imply the desired liveness property. The problem is to decompose this proof into a series of simple steps.

The decomposition is based upon the following observation. Let $\mathcal{A}$ be a well-founded set of predicates. Suppose that, using safety properties of the form $P \Rightarrow \Box R$, for every predicate A in $\mathcal{A}$ we can prove that

$$A \leadsto (Q \vee \exists A' \in \mathcal{A} : A \succ A')$$

The well-foundedness of $\mathcal{A}$ then implies that Q must eventually become true. This decomposition is indicated by a *proof lattice*[3] consisting of Q and the elements of $\mathcal{A}$ connected by lines, where downward lines from A to $A_1, \ldots,$ A_n denotes the assertion $A \leadsto A_1 \vee \ldots \vee A_n$.

An argument using a proof lattice $\mathcal{A}$ of predicates is completely equivalent to a counting argument using a function w with values in a well-founded set; either type of argument is easily translated into the other. These counting arguments work well for proving liveness properties that do not depend upon fairness assumptions. When fairness is required, it is convenient to use more general proof lattices containing arbitrary temporal logic formulas, not just predicates.

To illustrate the use of such proof lattices, we consider the mutual exclusion algorithm of Figure 2. For simplicity, the noncritical sections have been eliminated and the critical sections are represented by atomic actions, which are assumed not to modify x or y. Under the fairness assumption that a continuously enabled action must eventually be executed, this algorithm guarantees that the first process eventually enters its critical section. (However, the second process might remain forever in its **while** loop.) The proof that the algorithm satisfies the liveness property $at(\alpha) \leadsto at(\gamma)$ uses the proof lattice of Figure 3. The individual $\leadsto$ relations represented by the lattice are numbered and are explained below.

1. $at(\alpha) \leadsto (at(\beta) \wedge x)$ follows from the fairness assumption, since action α is enabled when $at(\alpha)$ is true.

2. This is an instance of the temporal logic tautology

$$P \leadsto (Q \vee (P \wedge \Box\neg Q))$$

   which is valid because Q either eventually becomes true or else remains forever false. (We are using linear-time temporal logic [Eme, section 2.3].)

[3]The term "proof lattice" is used even though $\mathcal{A}$ need not be a lattice.

```
variables x, y : boolean;
cobegin loop α: ⟨ x := true ⟩;
             β: ⟨ await ¬y ⟩;
             γ: ⟨ critical section ⟩;
                ⟨ x := false ⟩
        end loop
  □
        loop ⟨ y := true ⟩;
             while ⟨ x ⟩ do ⟨ y := false ⟩;
                         λ: ⟨ await ¬x ⟩;
                            ⟨ y := true ⟩
                         od
             ⟨ critical section ⟩;
             ⟨ y := false ⟩
        end loop
coend
```

Figure 2: A simple mutual exclusion algorithm.

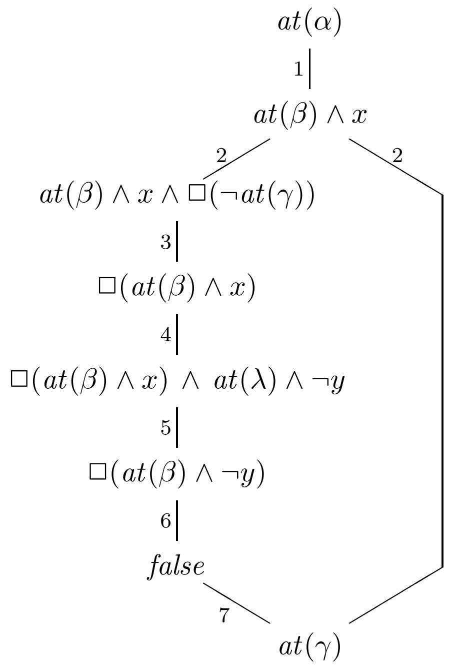

Figure 3: Proof lattice for mutual exclusion algorithm.

3. This $\leadsto$ relation is actually an implication, asserting that if the first process is at statement β with x true and never reaches γ, then it must remain forever at β with x true. This implication is of the form $(P \land \Box R) \Rightarrow \Box Q$ and is proved by finding an invariant under the constraint R, as explained in Section 3.4.1.

4. If x remains true forever, then the fairness assumption implies that control in the second process must eventually reach λ with y false. A formal proof of this assertion would use another proof lattice in which each $\leadsto$ relation represents a single step of the second process.

5. This is another property of the form $(P \land \Box R) \Rightarrow \Box Q$, proved by finding an invariant under the constraint R.

6. Action β is enabled when $at(\beta) \land \neg y$ holds, so by the fairness assumption, $\Box(at(\beta) \land \neg y)$ implies that β must eventually be executed, making $at(\beta)$ false. Since $\Box at(\beta)$ asserts that $at(\beta)$ is never false, this is a contradiction.

7. *false* implies anything.

The proof lattice formalizes a simple style of intuitive reasoning. Further examples of the use of proof lattices can be found in [OL82].

Temporal logic appears to be the best method for proving liveness properties that depend upon fairness assumptions. There seems little reason to use less formal methods for reasoning about behaviors, since such reasoning can be expressed compactly and precisely with temporal logic. However, the verification of liveness properties has received less attention than the verification of safety properties, and any conclusions we draw about the best approach to verifying liveness properties must be tentative.

3.5 Deriving Algorithms

We have discussed methods for reasoning about algorithms, without regard to how the algorithms are developed. There is increasing interest in methods for deriving correct algorithms. Exactly what is meant by "deriving" an algorithm varies. It may consist of simply developing the correctness proof along with the algorithm. Such an approach, based upon assertional methods and the Unity language, is taken by Chandy and Misra [CM88]. At the other extreme are approaches in which the program is derived automatically from a formal specification [Eme, section 7.3].

An appealing approach to the development of correct algorithms is by program transformation. One starts with a simple algorithm whose correctness is obvious, and transforms it by a series of refinement steps, where each step yields an equivalent program. Perhaps the most elegant instance of this approach is Milner's Calculus of Communicating Systems (CCS) [Mil80], where refinement steps are based upon simple algebraic laws. However, the simplicity and elegance of CCS break down in the presence of fairness, so CCS is not well suited for developing algorithms whose correctness depends upon fairness.

Methods for deriving concurrent algorithms are comparatively new and have thus far had only limited success. Automatic methods can derive only simple, finite-state algorithms. While informal methods can often provide elegant *post hoc* derivations of existing algorithms, it is not clear how good they are at deriving new algorithms. Finding efficient algorithms—whether efficiency is judged by theoretical complexity measures or by implementation in a real system—is still an art rather than a science. We still need to verify algorithms independently of how they are developed.

3.6 Specification

To determine whether an algorithm is correct, we need a precise specification of the problem it purports to solve. In the classical theory of computation, a problem is specified by describing the correct output as a function of the input. Such an input/output function is inadequate for specifying a problem in concurrency, which may involve a complex interaction of the system and its environment.

As discussed above, a behavior of a concurrent system is usually modeled as a sequence of states and/or actions. A specification of a system—that is, a specification of what the system is supposed to do—consists of the set of all behaviors considered to be correct. Another approach, taken by CCS [Mil80], is to model a concurrent system as a tree of possible actions, where branching represents nondeterminism. The specification is then a single tree rather than a set of sequences.

With any specification method, there arises the question of exactly what it means for a particular system to implement a specification. This is a very subtle question. Details that are insignificant for sequential programs may determine whether or not it is even possible to implement a specification of a concurrent system. Some of the issues that must be addressed are:

- No system can function properly in the face of completely arbitrary behavior by the environment. How can an implementation specify appropriate constraints on the environment (for example, that the environment not change the program's local variables) without "illegally" constraining the environment (for example, by preventing it from generating any input)?

- The granularity of action of the specification is usually much coarser than that of the implementation—for example, sending a message may be a single specification action, while executing each computer instruction is a separate implementation action. What does it mean to implement a single specification action by a set of lower-level actions?

- The granularity of data in the specification may be coarser than in the implementation—for example, messages versus computer words. What does it mean to implement one data structure with another?

Space does not permit a description of proposed specification methods and how they have addressed (or failed to address) these issues. We can only refer the reader to a small selection from the extensive literature on specification [LS84, Lam89, LT87, SM82].

4 Some Typical Distributed Algorithms

In this section, we discuss some of the most significant algorithms and impossibility results in this area. We restrict our attention to four major categories of results: shared variable algorithms, distributed consensus algorithms, distributed network algorithms and concurrency control. Although we are neglecting many interesting topics, these four areas provide a representative picture of distributed computing.

In early work, algorithms were presented rather informally, without formal models or rigorous correctness proofs. The lack of rigor led to errors, including the publication of incorrect algorithms. The development of formal models and proof techniques such as those discussed in Section 3, as well as a generally higher standard of rigor, has made such errors less common. However, algorithms are still published with inadequate correctness proofs, and synchronization errors are still a major cause of "crashes" in computer systems.

4.1 Shared Variable Algorithms

Shared variable algorithms represent the beginnings of distributed computing theory, and many of the ideas that are important elsewhere in the area first appear here. Today, programming languages provide powerful synchronization primitives and multiprocess computers provide special instructions to simplify their implementation, so the early synchronization algorithms are seldom used. However, higher-level contention and cooperation problems still exist, and these early algorithms provide insight into these problems.

4.1.1 Mutual Exclusion

The prototypical contention problem is that of *mutual exclusion*. Dijkstra [Dij65] presents a mutual exclusion algorithm which uses indivisible read and write operations on shared variables. In addition to ensuring mutual exclusion, the algorithm ensures the liveness property that some process eventually enters its critical section if there are any contending processes. Lockout freedom is not guaranteed; the system might grant the resource repeatedly to the same process, excluding another process forever. This algorithm is significant because prior to its discovery, it was not even clear that the problem could be solved.

Dijkstra's algorithm inspired a succession of additional solutions to the mutual exclusion problem. Some of this work improves upon his algorithm by adding the requirement that the solution be fair to individual processes. Fairness can take several forms. The strongest condition usually stated is FIFO (first-in first-out), while the weakest is lockout freedom. There are intermediate possibilities: there might be an upper bound on the number of times one process can be bypassed by another while it is waiting for the resource ("bounded waiting"), or, the time for a process to obtain the resource might be bounded in terms of its own step time. (These last two conditions are very different: the former is an egalitarian condition which tends to cause all processes to move at the same speed, while the latter tends to allow faster processes to move ahead of slower processes.) The work on mutual exclusion includes a collection of algorithms satisfying these various fairness conditions.

An interesting example of a mutual exclusion algorithm is Lamport's "bakery algorithm" [Lam74], so called because it is based on the processes choosing numbers, much as customers do in a bakery. The bakery algorithm was the first FIFO solution, and it was the first solution to use only local shared variables (see Section 2.2.1). It also has the fault-tolerance property that if a process stops during its protocol, and its local shared variables subsequently revert to their initial values, then the rest of the system continues correctly without it. This property permits a distributed implementation that tolerates halting failures.

The most important property of the bakery algorithm is that it was the first algorithm to implement mutual exclusion without assuming lower-level mutual exclusion of read and write accesses to shared variables. Accesses to shared variables may occur concurrently, where reads that occur concurrently with writes are permitted to return arbitrary values. Concurrent reading and writing is discussed in Section 4.1.4.

Peterson and Fischer [PF77] contribute a complexity-theory perspective to the mutual exclusion area. They describe a collection of algorithms which include strong fairness and resiliency properties, and which also keep the size of the shared variables small. Of particular interest is their "tournament algorithm", which builds an n-process mutual exclusion algorithm from a binary tree of 2-process mutual exclusion algorithms. They also describe a useful way to prove bounds on time complexity for asynchronous parallel algorithms: assuming upper bounds on the time for certain primitive occurrences (such as process step time and time during which a process holds the resource), they infer upper bounds on the time for occurrences of interest (such as the time for a requesting process to obtain the resource). Their method can be used to obtain reasonable complexity bounds, not only for mutual exclusion algorithms, but also for most other types of asynchronous algorithms.

The development of many different fairness and resiliency conditions, and of many complex algorithms, gave rise to the need for rigorous ways of reasoning about them. Burns et al. [BJL*82] introduce formal models for shared-variable algorithms, and use the models not only to describe new memory-efficient algorithms, but also to prove impossibility results and complexity lower bounds. The upper and lower bound results in [BJL*82] are for the amount of shared memory required to achieve mutual exclusion with various fairness properties. The particular model assumed there allows for a powerful sort of access to shared memory, via indivisible "test and set" (combined read and write) operations. Even so, Burns and his coauthors are able to prove that $\Omega(n)$ different values of shared memory are required to guarantee fair mutual exclusion. More precisely, guaranteeing freedom from lockout requires at least $n/2$ values, while guaranteeing bounded waiting requires at least n values.

The lower bound proofs in [BJL*82] are based on the limitations of "local knowledge" in a distributed system. Since processes' actions depend only on their local knowledge, processes must act in the same way in all computations that look identical to them. The proofs assume that the shared memory has fewer values than the claimed minimum and derive a contradiction. They do this by describing a collection of related computations and then using the limitation on shared memory size and the pigeonhole principle to conclude that some of these computations must look identical to certain processes. But among these computations are some for which the problem specification requires the processes to act in different ways, yielding a contradiction. The method used here—proving that actions based on local knowledge can force two processes to act the same when they should act differently—is the fundamental method for deriving lower bounds and other impossibility results for distributed algorithms.

The lower bound results in [BJL*82] apply only to deterministic algorithms—that is, algorithms in which the actions of each process are uniquely determined by its local knowledge. Recently, randomized algorithms, in which processes are permitted to toss fair coins to decide between possible actions, have emerged as an alternative to deterministic algorithms. A randomized algorithm can be thought of as a strategy for "playing a game" against an "adversary", who is usually assumed to have control over the inputs to the algorithm and the sequence in which the processes take steps. In choosing its own moves, the adversary may use knowledge of previous moves. A randomized algorithm should, with very high probability, perform correctly against any allowable adversary.

One of the earliest examples of such a randomized algorithm was developed by Rabin [Rab82] as a way of circumventing the limitations proved in [BJL*82]. The shared memory used by Rabin's algorithm has only $O(logn)$ values, in contrast to the $\Omega(n)$ lower bound for deterministic algorithms. Rabin's algorithm is also simpler than the known deterministic mutual exclusion algorithms that use $O(n)$-valued shared memory. A disadvantage is that Rabin's algorithm is not solving exactly the same problem—it is not absolutely guaranteed to grant the resource to every requesting process. Rather, it does so with probability that grows with the amount of time the process waits. Still, in some situations, the advantages of simplicity and improved performance may outweigh the small probability of failure.

The mutual exclusion problem has also been studied in message-passing models. The first such solution was in [Lam78], where it was presented as a simple application of the use of logical clocks to totally order system events (see Section 2.3). Mutual exclusion was reduced to the global consistency problem of getting all processes to have a consistent view of the queue of waiting processes. More recently, several algorithms have been devised which attempt to limit the number of messages required to solve the problem. A generalization to *k-exclusion*, in which up to k processes can be in their critical section at the same time has also been studied.

The reader can consult the book by Raynal [Ray86] for more information and more pointers into the extensive literature on mutual exclusion.

4.1.2 Other Contention Problems

The *dining philosophers* problem [Dij71] is an important resource allocation problem in which each process ("philosopher") requires a specific set of resources ("forks"). In the traditional statement of the problem, the philosophers are arranged in a circle, with a fork between each pair of philosophers. To eat, each philosopher must have both adjacent forks. Dijkstra's solution is based on variables (semaphores) shared by all processes, and thus is best suited for use within a single computer.

One way to restrict access to the shared variables is by associating each variable with a resource, and allowing only the processes that require that resource to access the variable. This arrangement suggests solutions in which processes simply visit all their resources, attempting to acquire them one at a time. Such a solution permits deadlock, where processes obtain some resources and then wait forever for resources held by other processes. In the circle of dining philosophers, deadlock arises if each one first obtains his left fork and then waits for his right fork.

The traditional dining philosophers problem is symmetrical if processes are identical and deterministic and all variables are initialized in the same way. If processes take steps in round-robin order, the system configuration is symmetrical after every round. This implies that, if any process ever obtained all of its needed resources, then every process would, which is impossible. Hence, there can be no such completely symmetric algorithm. The key to most solutions to this problem is their method for breaking symmetry.

There are several ways of breaking symmetry. First, there can be a single "token" that is held by one process, or circulated around the ring. To resolve a conflict, the process with the token relinquishes its resources in exchange for a guarantee that it can have them when they next become available. Second, alternate processes in an even-sized ring can attempt to obtain their left or right resources first; this strategy can be used not only to avoid deadlock, but also to guarantee a small upper bound on waiting time for each process. Third, Chandy and Misra [CM84] describe a scheme in which each resource has a priority list, describing which processes have stronger claims on the resource. These priorities are established dynamically, depending on the demands for the resources. Although the processes are identical, the initial configuration of the algorithm is asymmetric: it includes a set of priority lists that cannot induce cycles among waiting processes. The rules used in [CM84] to modify the priority lists preserve acyclicity, and so deadlock is avoided.

Finally, Rabin and Lehmann [RL81] describe a simple randomized algorithm that uses local random choices to break symmetry. Each process chooses randomly whether to try to obtain its left or right fork first. In either case, the process waits until it obtains its first fork, but only tests once to see if its second fork is available. If it is not, the process relinquishes its first fork and starts over with another random choice. This strategy guarantees that, with probability 1, the system continues to make progress.

These symmetry-breaking techniques avoid deadlock and ensure that the system makes progress. They provide a variety of fairness and performance guarantees.

4.1.3 Cooperation Problems

For shared-variable models, cooperation problems have received less attention than contention problems. The only cooperation problems that have been studied at any length are *producer-consumer* problems, in which processes produce results that are used as input by other processes. The simplest producer-consumer problem is the bounded buffer problem (Section 2.3). A very general class of producer-consumer problem involves the simulation of a class of Petri nets known as marked graphs [CHEP71], where each node in the graph represents a process and each token represents a value. An example of this class is the problem of passing a token around a ring of processes, where the token can be used to control access to some resource.

An interesting problem that combines aspects of both contention and cooperation is concurrent garbage collection, in which a “collector” process running asynchronously with a “mutator” process must identify items in the data structure that are no longer accessible by the mutator and add those items to a “free list”. This is basically a producer-consumer problem, with the collector producing free-list items and the mutator consuming them. However, the problem also involves contention because the mutator changes the data structure while the collector is examining it.

In shared-variable models, cooperation problems have not been studied as extensively as contention problems, probably because they are easier to solve. For example, in concurrent garbage collection algorithms, it is the contention for access to the data structure rather than the cooperative use of the free list that poses the challenge. However, there is one important property that is harder to achieve in cooperation problems than in contention problems—namely, *self-stabilization.* An algorithm is said to be self-stabilizing if, when started in any arbitrary state, it eventually reaches a state in which it operates normally [Dij74]. For example, a self-stabilizing token-passing algorithm can be started in a state having any number of tokens and will eventually reach a state with just one token that is being passed around. It is generally easy to devise self-stabilizing contention problems because processes go through a "home" state in which they are reinitialized—for example, a process in the dining philosopher problem eventually reaches a state in which it is not holding or requesting any forks—and the whole algorithm is reinitialized when every process has reached its home state. On the other hand, cooperation problems do not have such a home state. For example, the symmetry in the bounded buffer problem means that an empty buffer and a full buffer are symmetric situations, and neither of them can be considered a "home" state. Dijkstra's self-stabilizing token-passing algorithms [Dij74] are currently the only published self-stabilizing cooperation algorithms.

Self-stabilization is an important fault-tolerance property, since it permits an algorithm to recover from any transient failure. This property has not received the attention it deserves.

4.1.4 Concurrent Readers and Writers

With the exception of the bakery algorithm, all of the work we have described so far assumes that processes access shared memory using primitive operations (usually read and write operations), each of which is executed indivisibly. The ability to implement multiple processors with a single integrated circuit has rekindled interest in shared memory models that do not assume indivisibility of reads and writes. Rather, they assume that operations on a shared variable have duration, that reads and writes that do not overlap behave as if they were indivisible, but that reads and writes that overlap can yield less predictable results [Lam86]. The bakery algorithm assumes *safe* shared variables—ones in which a read that is concurrent with a write can return an arbitrary value from the domain of possible values for the variable. Another possible assumption is a *regular* shared variable, in which a read that overlaps a write is guaranteed to return either the old value or the one being written; however, two successive reads that overlap the same write may obtain first the new value then the old one. A still stronger assumption is an *atomic* shared variable, which behaves as if each read and each write occurred at some fixed time within its interval.

Using safe, regular, or atomic shared variables, it is possible to simulate shared variables having indivisible operations, so that algorithms designed for the stronger models can be applied in the weaker models. This work has evolved from the traditional readers-writers algorithms based on mutual exclusion [CHP71], through nontraditional algorithms that allow concurrent reading and writing [Pet83], to more recent algorithms for implementing one class of shared variable with a weaker class [Lam86, BP87, Blo88].

Recently, Herlihy [Her88] has considered atomic shared variables that support operations other than reads and writes. He has shown that read-write atomic variables cannot be used to implement more powerful atomic shared variables such as those supporting test-and-set operations. He has also shown that other types of atomic variables are "universal", in the sense that they can be used to implement atomic shared variables of arbitrary types. Herlihy's impossibility proof proceeds by showing that atomic read-write shared variables cannot be used to solve a version of the distributed consensus problem discussed in the following subsection.

4.2 Distributed Consensus

Achieving global consistency requires that processes reach some form of agreement. Problems of reaching agreement in a message-passing model are called *distributed consensus* problems. There are many such problems, including agreeing (exactly or approximately) on values from some domain, synchronizing actions of different processes, and synchronizing software clocks. Distributed consensus problems arise in areas as diverse as real-time process-control systems (where agreement might be needed on the values read by replicated sensors) and distributed database systems (where agreement might be needed on whether or not to accept the results of a transaction). Since global consistency is what makes a collection of processes into a single system, distributed consensus algorithms are ubiquitous in distributed systems.

Consensus problems are generally easy to solve if there are no failures; in this case, processes can exchange information reliably about their local states, and thereby achieve a common view of the global state of the system. The problem is considerably harder, however, when failures are considered. Consensus algorithms have been presented for almost all the classes of failure described in Section 2.1.1.

Distributed consensus problems have been a popular subject for theoretical research recently, because they have simple mathematical formulations and are surprisingly challenging. They also provide a convenient vehicle for comparing the power of models that make different assumptions about time and failures.

4.2.1 The Two-Generals Problem

Probably the first distributed consensus problem to appear in the literature is the "two-generals problem" [Gra78], in which two processes must reach agreement when there is a possibility of lost messages. The problem is phrased as that of two generals, who communicate by message, having to agree upon whether or not to attack a target. The following argument can be formalized to show that the problem is unsolvable when messages may be lost. Reaching at least one of the two possible decisions, say the decision to attack, requires the successful arrival of at least one message. Consider a scenario Σ in which the fewest delivered messages that will result in agreement to attack are delivered, and let Σ' be the same scenario as Σ except that the last message delivered in scenario Σ is lost in Σ', and any other messages that might later be sent are also lost. Suppose this last message is from general A to general B. General A sees the same messages in the two scenarios, so he must decide to attack. However, the minimality assumption of Σ implies that B cannot also decide to attack in scenario Σ', so he must make a different decision. Hence, the problem is unsolvable.

4.2.2 Agreement on a Value

The agreement problem requires that processes agree upon a value. Communication is assumed to be reliable, but processes are subject to failures (either halting, omission, or Byzantine). Each of the processes begins the algorithm with an input value. After the algorithm has completed, each process is to decide upon an output value. There are two constraints on the solution: (a) (Agreement) all nonfaulty processes must agree on the output, and (b) (Validity) if all nonfaulty processes begin with the same input value, that value must be the output value of all nonfaulty processes. For the case of Byzantine faults, this problem has been called the *Byzantine generals problem*. (Other, equivalent formulations of the problem have also been used.)

In the absence of failures, this problem is easy to solve: processes could simply exchange their values, and each could decide upon the majority value. The following example shows the kinds of difficulties that can occur, when failures are considered. Consider three processes, A, B and C. Suppose that A and B begin with input 0 and 1 respectively. Suppose that C is a Byzantine faulty processor, which acts toward A as if C were nonfaulty and started with 0, but as if B were faulty. At the same time, C acts toward B as if C were nonfaulty and started with 1, but as if A were faulty. Since A's view of the execution is consistent with A and C being nonfaulty and starting with the same input, 0, A is required to decide 0. Analogously, B is required to decide 1. But this means that A and B have been made to disagree, violating the agreement requirement of the problem. This example can be elaborated into a proof of the impossibility of reaching agreement among $3t$ processes if t processes might be faulty.

The problem of reaching agreement on a value was studied by Pease, Shostak, and Lamport [PSL80, LSP82] in a model with Byzantine failures and computation performed in a sequence of rounds. (They also described the implementation of rounds with synchronized clocks.) Besides containing the impossibility proof described in the last paragraph, these papers also contain two subtle algorithms. The first is a recursive algorithm that requires $3t+1$ processes and tolerates Byzantine faults. The second requires only $t+1$ processes, but assumes digital signatures (Section 2.1.1). Both algorithms assume a completely connected network.

Dolev [Dol82] considers the same problem in an arbitrary network graph. For t Byzantine failures, he shows how to implement an algorithm similar to that of [LSP82], provided that the network is at least $2t + 1$-connected (and has at least $3t + 1$ processes). He also proves a matching lower bound.

A series of results, starting with [FL81] and culminating in [DM86], shows that any synchronous algorithm for reaching agreement on a value, in the presence of t failures—even the simple halting failures—requires at least $t + 1$ rounds of message exchange in the worst case. As usual, these arguments are based on the limitations caused by local knowledge in distributed algorithms; by assuming fewer rounds, a "chain" of computations is constructed that leads to a contradiction. In the first computation in the chain, nonfaulty processes are constrained by the problem statement to decide 0, while in the last computation in the chain, nonfaulty processes are constrained to decide 1. Further, any two consecutive computations in the chain share a nonfaulty process to which the two computations look the same; this process therefore reaches the same decision in both computations. Hence, all nonfaulty processes decide upon the same value in every computation in the chain, which yields the required contradiction.

Dwork and Moses [DM86] provide explicit, intuitive definitions for the "knowledge" that individual processes have at any time during the execution of an algorithm. Their problem statements, algorithms, and lower bound proofs are based on these definitions. This work suggests that formal models and logics of knowledge may provide useful high-level ways of reasoning about distributed algorithms.

Bracha [Bra85] is able to circumvent the $t + 1$ lower bound on rounds with a randomized algorithm; his solution uses only $O(logn)$ rounds, but requires cryptographic techniques that rest on special assumptions. More recently, Feldman and Micali [FM88] have improved Bracha's upper bound to a constant. Chor and Coan [CC84] give another randomized algorithm that requires $O(t/logn)$ rounds, but does not require any special assumptions.

The consensus algorithms mentioned above all assume a synchronous model of computation. Fischer, Lynch, and Paterson [FLP85] study the problem of reaching agreement on a value in a completely asynchronous model. They obtain a surprising fundamental impossibility result: if there is the possibility of even one simple halting failure, then an asynchronous system of deterministic processes cannot guarantee agreement. This result suggests that, while asynchronous models are simple, general, and popular, they are too weak for studying fault tolerance.

The impossibility result is proved by first showing that any asynchronous consensus protocol that works correctly in the absence of faults must have a reachable configuration C in which there is a single "decider" process i—one that is capable, on its own, of causing either of two different decisions to be reached. If this protocol is also required to tolerate a single process failure, then, starting from C, all the processes except i must be able to reach a decision. But, this decision will conflict with one of the possible decisions process i might reach on its own. (Herlihy used a similar technique to prove the impossibility result mentioned in the previous subsection.)

There are several ways to cope with the limitation described in [FLP85].

One can simply add some synchrony assumptions—the weakest ones commonly used are timers and bounded message delay. Alternatively, one can use an asynchronous deterministic algorithm, but attempt to reduce the probability that a failure will upset correct behavior. This approach is sometimes used in practice when only modest reliability is needed, but there has been no rigorous attempt to analyze the reliability of the resulting system.

Another possibility is to use randomized rather than deterministic algorithms. For example, Ben-Or [Ben83] gives a randomized algorithm for reaching agreement on a value in the completely asynchronous model, allowing Byzantine faults. The algorithm never permits disagreement or violates the validity condition; however, instead of guaranteeing eventual termination, it guarantees only termination with probability 1.

A good survey of the early work in this area appears in [Fis83].

4.2.3 Other Consensus Problems

Other distributed consensus problems have been studied under the assumption that processes can be faulty but communication is reliable. One such problem is that of reaching approximate, rather than exact, agreement on a value. Each process begins with an initial (infinite-precision) real value, and must eventually decide on a real value subject to: (a) (Agreement) all nonfaulty processes' decisions must agree to within ϵ, and (b) (Validity) the decision value for any nonfaulty process must be within ϵ of the range of the initial values of the nonfaulty processes. Processes are permitted to send real values in messages.

Although the problems of exact and approximate agreement seem to be quite similar, reaching approximate agreement is considerably easier; in particular, there are simple deterministic algorithms for approximate agreement in asynchronous models—even in the presence of Byzantine faults. It seems almost paradoxical that deterministic processes can reach agreement on real values to within any predetermined ϵ, but they cannot reach exact agreement on a single bit.

Another consensus problem is achieving simultaneous action by distributed processes, in a model with timers in which all messages take exactly the same (known) time for delivery. This problem, sometimes called the "distributed firing squad problem", yields results very similar to those for agreement on a value. In fact, for the case of Byzantine faults, a general transformation converts any algorithm for agreement to an algorithm for simultaneous action. The firing squad algorithm is obtained by running many instances of the agreement algorithm, each deciding whether the processes should fire at a particular time. The first instance that reaches a positive decision triggers the simultaneous firing action.

In this transformation, many instances of a Byzantine agreement algorithm are executed concurrently. Those instances that are not actually carrying out any interesting computation can be implemented in a trivial way by letting all of their messages be special "null" messages that are not actually sent. This trick of sending a message by not sending a message is also used in [Lam84] to give fault-tolerant distributed simulations of centralized algorithms.

Another consensus problem is establishing and maintaining synchronized local clocks in a distributed system. It is closely related to both of the preceding problems (reaching approximate agreement and achieving simultaneous action), since it may be viewed as simultaneously reaching approximate agreement on a clock value, or as reaching exact agreement on a clock value at approximately the same instant. The problem is one of implementing synchronized clocks using timers that run at approximately the same rate, usually assuming initial synchronization of the clocks. However, it is generally described in terms of maintaining the synchronization (to within ϵ) of the processes' clocks despite a small, varying difference in their clock rates.

Clock synchronization is difficult to achieve in the presence of faulty processes. Many algorithms to solve this problem have been suggested, analyzed, and compared in the literature, and some have been used in implementing systems. In most algorithms for maintaining synchronization among clocks that are initially synchronized, a new round is begun when the clocks reach predetermined values. In each round, processes exchange information about their clock values and use the information to adjust their own clocks. Synchronization algorithms that do not assume the clocks to be initially synchronized use other methods, since they cannot depend upon the clocks to determine when the first round should begin.

Lower bounds and impossibility results have also been proved for clock synchronization problems. Of particular interest is the result of Dolev, Halpern, and Strong [DHS84] showing that clock synchronization problems cannot be solved for $3t$ processes if t of them can exhibit Byzantine failures.

This impossibility result is reminiscent of the impossibility result described earlier for agreement on a value, where the problem cannot be solved with $3t$ processes in the presence of t Byzantine failures. In fact, a $3t$ versus t impossibility result also holds for many other consensus problems under Byzantine failures, including approximate agreement and simultaneous ac-

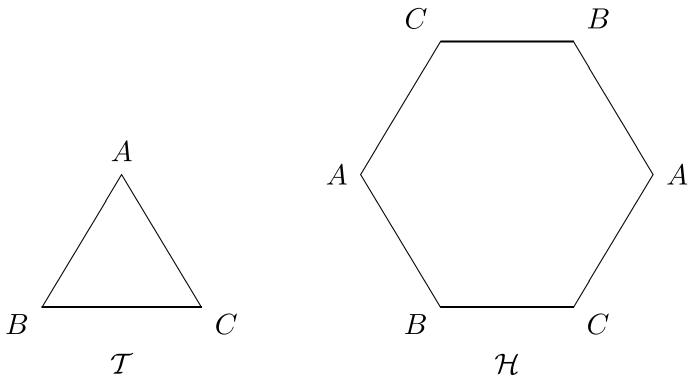

Figure 4: The systems $\mathcal{T}$ and $\mathcal{H}$

tion. Furthermore, all of these problems are unsolvable in network graphs having less than $2t+1$-connectivity. These impossibility results do not apply if authentication is used.

Since all of these bounds are tight, it is apparent that there must be a common reason for the many similar results. Fischer, Lynch, and Merritt [FLM86] tie together this large collection of impossibility results with a common proof technique. We illustrate this technique by proving the 3-versus-1 impossibility result for reaching agreement on a value. Assume for the sake of obtaining a contradiction that there is such a solution for the system $\mathcal{T}$ consisting of the three processes A, B, and C arranged in a triangle. Let $\mathcal{H}$ be a new system, consisting of two copies of each of A, B and C, in the hexagonal arrangement shown in Figure 4. Note that system $\mathcal{H}$ looks locally like the original system $\mathcal{T}$.

Let Σ be a computation of $\mathcal{H}$ that results if $\mathcal{H}$ is run with each of its six processes behaving exactly like the corresponding nonfaulty process of $\mathcal{T}$. Consider any pair of nonfaulty processes in $\mathcal{H}$, say the upper-right-hand copies of A and B. There is a computation Σ' of $\mathcal{T}$, with C faulty, in which A and B receive the same inputs as they do in the computation Σ. By our assumption, A and B agree on the same value in Σ'. Since the copies of A and B have the same view in Σ as their namesakes do in Σ', they must also agree on the same value. Moreover, if A and B have the same input value, then that is their output value. Since this proof works for any pair of adjacent processes in $\mathcal{H}$, this shows that in any computation of $\mathcal{H}$: (a) all processes must agree upon the same output value, and (b) if adjacent processes have the same input value then that is their output value. Letting the upper-right-hand copies of A and B have input values of 1 and the lower-left-hand copies of A and B have input values of 0, this implies that the output of every process must equal both 0 and 1, which is the required contradiction. Other impossibility results are proved similarly, using slightly more complicated systems for $\mathcal{H}$.

4.2.4 The Distributed Commit Problem

The *transaction commit* problem for distributed databases is the problem of reaching agreement, among the nodes that have participated in a transaction, about whether to *commit* or *abort* the transaction. (We say more about transactions in Section 4.4.) The requirements are: (a) (Agreement) all nonfaulty processes' decisions must agree, and (b) (Validity) (i) if any process's initial value is "abort", then the decision must be "abort", and (ii) if all processes' initial values are "commit" and no failure occurs, then the decision must be "commit". The problem has traditionally been studied under the assumption of halting failures and the loss of individual messages. The impossibility result of [FLP85] implies that the commit problem cannot be solved in the completely asynchronous model, for even a single faulty process—even with reliable communication. The impossibility result for the two-generals problem implies that the commit problem cannot be solved if messages can be lost, even if message delays are otherwise bounded and processes are reliable and have synchronized clocks.

Most commit protocols, such as the popular *two-phase commit* algorithm, have a failure window—a period during the computation when a single halting failure can prevent termination. Using the assumptions that the processes have synchronized clocks and there is a known upper bound on message delivery time, one can construct a commit protocol that has no failure window from a synchronous algorithm for reaching agreement on a value. However, the synchronous model does not permit communication failure, so the loss of a message must be considered to be a failure of either the sending or receiving process. The *three-phase commit* protocol of Skeen [Ske82] is another commit protocol without a failure window; it assumes reliable message delivery and detectable failures.

4.3 Network Algorithms

We now describe a class of algorithms for message-passing models, which we call network algorithms, in which the behavior of the algorithm depends strongly on the network topology. Most of these algorithms are designed to solve problems arising in communication in computer networks. They usually assume a completely asynchronous, failure-free model. Most of them can be divided into two categories, which we call *static* and *dynamic*. Static algorithms are assumed to operate in fixed networks and to start with all their inputs available at the beginning; dynamic algorithms also operate in fixed networks but receive some of their inputs interactively. Another way of viewing the distinction is that static algorithms are based upon unchanging information in the initial states of the processes, while dynamic algorithms use changing information from the changing state of the application processes. A network problem can have both static and dynamic versions, but the two versions are usually treated separately in the literature. We also consider some algorithms designed to operate in changing networks, and some algorithms designed to ensure reliable message delivery over a single unreliable link.

4.3.1 Static Algorithms

Route-Determination Algorithms In communication networks, it is often important for processes that have local information about the speed, bandwidth, and other costs of message transmission to their immediate network neighbors, to determine good routes through the network for communicating with distant processes. If such routes are to be determined infrequently, it may be useful to consider the static problem in which the local information is assumed to be available initially and fixed during execution of the route-determination algorithm.

Different applications require different notions of what constitutes a "good" set of routes through the network. For example, if the routes are used primarily for broadcasting a single message to all other processes, unnecessary message duplication can be avoided by establishing a spanning tree of the network. If a weight is associated with each link in the network to represent the cost of sending a message over that link, a minimum-weight spanning tree (*MST*) can be used to minimize the total cost of the broadcast.

Gallager, Humblet, and Spira [GHS83] present an efficient distributed algorithm for finding a minimum-weight spanning tree in a network with n

nodes and e edges. The algorithm is based upon the following two observations: if all edge weights are distinct, then the MST is unique; and the minimum-weight external edge of any subtree of the MST is in the MST. The algorithm grows the MST by coalescing fragments until the complete MST is formed. Initially, each node is a fragment, and a fragment coalesces with the one at the other end of its minimum-weight external edge.

The main achievement of this algorithm is to keep the number of messages small. Each time a fragment with f nodes computes its minimum-weight external edge, $O(f)$ messages are required. Naively coalescing fragments could lead to as many as $\Omega(n^2)$ messages. By using a priority scheme to determine when fragments are permitted to coalesce, this algorithm generates only $O(nlogn + e)$ messages.

Although the basic idea is simple, the algorithm itself is quite complicated. Certain simple "high-level" tasks, such as determining a fragment's minimum-weight external edge, are implemented as a series of separate steps occurring at different processes. The steps implementing different high-level tasks interleave in complicated ways. The correctness of the algorithm is not obvious; in fact, only recently have careful correctness proofs appeared. [GC88, WLL88] While these proofs use techniques based upon those described in Section 3, they are lengthy and difficult to check. In general, network algorithms are typically longer and harder to understand than the other types of distributed algorithms we are considering, and rigorous correctness proofs are seldom given.

Many other network algorithms are also designed to minimize the number of messages sent. While message complexity is easy to define and amenable to clean upper and lower bound results, time bounds may be more important in practice. However, there have so far been few upper and lower time bounds derived for network problems.

Other route-determination algorithms have been proposed for finding MST's in a directed graph of processes [Hum83] and for determining other routing structures, such as the set of shortest paths between all pairs of nodes and breadth-first and depth-first spanning trees. Also, a basic lower bound of $\Omega(e)$ has been proved for the number of messages required to implement broadcast in an arbitrary network [AGPV88].

Leader Election In this problem, a network of identical processes must choose a "leader" from among themselves. The processes are assumed to be indistinguishable, except that they may possess unique identifiers. The difficulty lies in breaking the symmetry. Solutions can be used to implement a fault-tolerant token-passing algorithm; if the token is lost, the leader-election algorithm is invoked to decide which process should possess the token.

Peterson [Pet82] has devised a leader-election algorithm for a completely asynchronous ring of processes with unidirectional communication; it uses at most $O(n \log n)$ messages in the worst case. On the other hand, Frederickson and Lynch [FL87] have shown that at least $\Omega(n \log n)$ messages are required in the worst case, even in a ring having synchronous and bidirectional communication.

These results would characterize the message complexity in the important special case of a ring of processes but for an interesting technicality. The Frederickson–Lynch lower bound assumes that the algorithm uses process identifiers only in order comparisons, but not in counting or more general arithmetic operations. Almost all published election algorithms satisfy this assumption. The lower bound also holds for more general uses of identifiers if, for each ring size, the algorithm satisfies a uniform time bound, independent of the process identifiers. Without this technical assumption, the problem can be solved with only $O(n)$ messages by an algorithm taking an unbounded amount of time [FL87, Vit84]. Although unlikely to be of practical use, this algorithm provides an interesting extreme time-message tradeoff.

The election problem has been solved under many different assumptions: the network can be a ring, a complete graph, or a general graph; the graph can be directed or undirected; the processes might have unique identifiers or be identical; the individual processes might or might not know the size or shape of the network; the algorithm can be deterministic or randomized; communication can be synchronous, asynchronous, or partially synchronous; and failures might or might not be allowed. The problem has provided an opportunity to study a single problem under many different assumptions, but no general principles have yet emerged.

Other Problems Other static problems include the computation of functions, such as the median and other order statistics, where the inputs are initially distributed. Attiya et al. [ASW88] and Abrahamson et al. [AAHK86] have obtained especially interesting upper and lower bound results—many surprisingly tight—about the number of messages required to compute functions in a ring.

4.3.2 Dynamic Algorithms

Distributed Termination In this problem, each process is either active or inactive. Only an active process can send a message, and an inactive process can become active only by receiving a message. A termination detection algorithm detects when no further process activity is possible—that is, when all processes are simultaneously inactive and no messages are in transit.

This problem was first solved by Dijkstra [DS80] for the special case in which the application program is a "diffusing computation"—one where all activity originates from and returns to one controlling process. Other researchers have addressed the problem of detecting termination in CSP programs; because CSP programs admit the possibility of deadlock as well as normal termination, these algorithms must also recognize deadlock. Termination can also be detected using global snapshot algorithms, discussed later in this section.

Distributed Deadlock Detection Here, it is assumed that processes request resources and release them, and there is some mechanism for granting resources to requesting processes. However, resources may be granted in such a way that deadlock results—for example, in the dining philosophers problem, each philosopher may have requested both forks and received only his right fork, so the system is deadlocked because no one can obtain his left fork. A deadlock detection algorithm detects such a situation, so appropriate corrective action can be taken—usually forcing some processes to relinquish resources already granted.

The simplest instance of the problem, in which each process is waiting for a single resource held by another process, is solved by detecting cycles of the form "A is waiting for a resource held by B, who is waiting for a resource ... held by A." Straightforward cycle-detection algorithms can be applied, but they may not be efficient. A more complicated solution is required if process requests have a more interesting structure, such as "any one of a set of resources" or "any two from set S and any one from set T". In such cases, the problem may involve detecting a knot or other graph structure, instead of a cycle.

A difficulty in designing distributed deadlock detection algorithms is avoiding the detection of "false deadlocks". Consider a ring of processes each of which occasionally requests a resource held by the next, but in which there is no deadlock. An algorithm that simply checks the status of all processes in some order could happen to observe every process when it is waiting for a resource and incorrectly decide that there is deadlock. The algorithm of Chandy, Misra, and Haas [CHM83] is a typical algorithm that does not detect false deadlocks.

Global Snapshots The global state of a distributed system consists of the state of each process and the messages on each transmission line. A global snapshot algorithm attempts to determine the global state of a system. A trivial algorithm would instantaneously "freeze" the execution of the system and determine the state at its leisure, but such an algorithm is seldom feasible. Moreover, as explained in Section 2.3, determining the global state requires knowing the complete temporal ordering of events, which may be impossible. Therefore, a global snapshot algorithm is required only to determine a global state that is consistent with the known temporal ordering of events. This is sufficient for most purposes. This problem was studied by Chandy and Lamport [CL85], who presented a simple global snapshot algorithm.

A global snapshot algorithm can be used whenever one wants global information about a distributed system. In a distributed banking system, such an algorithm can determine the total amount of money in the bank without halting other banking transactions. Similarly, one can use a global snapshot algorithm to checkpoint a system for failure recovery without halting the system.

A general class of applications is detecting when an invariant property holds. Recall that an invariant is a property of the state which, once it holds, will continue to hold in all subsequent states. (Invariants of distributed systems are often called "stable properties".) Distributed termination and deadlock are invariants. If an invariant holds in the consistent state observed by a global snapshot algorithm, then it also holds in all global states reached by the system after the snapshot algorithm terminates. Thus, one can detect termination, deadlock, or any other invariant property by obtaining a consistent global snapshot and checking it for that property.

A global snapshot algorithm can transform an algorithm for solving a static network problem to one that solves the dynamic version of the same problem. For example, the static version of the deadlock detection problem, in which the set of resources held and requests pending never changes, is easier to solve than the dynamic version because there is no possibility of detecting false deadlocks. The harder dynamic version can be solved by taking a global snapshot, then running an algorithm for the static problem on the state determined by that snapshot. It is not necessary to collect the global snapshot information in one place; the static deadlock detection can be done with a distributed algorithm. This strategy is used in a deadlock detection algorithm by Bracha and Toueg [BT87].

Synchronizers Many simple algorithms have been designed for strongly synchronous networks—ones in which the entire computation proceeds in a series of rounds. A *network synchronizer* is a program designed to convert such an algorithm to one that can run in a completely asynchronous network. Awerbuch [Awe85] has designed a collection of network synchronizers, varying in their message and time complexity. They have been used to produce asynchronous algorithms that are more efficient than previously known ones for breadth-first search and the determination of maximum flows and shortest paths.

The simplest of the synchronizers transforms an algorithm for a synchronous network into an asynchronous algorithm that has approximately the same execution time. This seems to imply that any problem can be solved just as quickly in asynchronous networks as in synchronous networks. However, Arjomandi, Fischer, and Lynch [AFL83] showed that there are some problems whose solution requires much more time (greater by a multiplicative factor of the network diameter) in an asynchronous than in a synchronous network. A typical problem is for all nodes to perform a sequence of outputs, in such a way that every node does its i^{th} output before any node does its $(i+1)^{\text{st}}$. A synchronous system can perform r such output rounds in time r, but an asynchronous system requires extra time for communication between all the nodes in between each pair of rounds.

4.3.3 Changing Networks

The algorithms discussed so far in this subsection are designed to operate in communication networks that are fixed while the algorithm is being executed. Algorithms for the same problems are also required for the harder case where network links may fail and recover during execution—that is, for *changing networks*.

One can translate any algorithm for a fixed but arbitrary network into one that works for a changing network as follows. The nodes run the fixed-network algorithm as long as the network does not appear to change. Whenever a node detects a change, it stops executing the old instance of the fixed-network algorithm and begins a new instance, this time on the changed network. Thus, there can be many instances of the fixed-network algorithm executing simultaneously. The different instances are distinguished by means of "instance identifiers" attached to the messages.

It is not difficult to implement this approach using an unbounded number of instance identifiers, each chosen to be larger than the previous one used by the node. Afek, Awerbuch, and Gafni [AAG87] have developed a method that requires only a finite number of identifiers. However, simply bounding the number of instance identifiers is of little practical significance, since practical bounds on an unbounded number of identifiers are easy to find. For example, with 64-bit identifiers, a system that chooses ten per second and was started at the beginning of the universe would not run out of identifiers for several billion more years. However, through a transient error, a node might choose too large an identifier, causing the system to run out of identifiers billions of years too soon—perhaps within a few seconds. A self-stabilizing algorithm using a finite number of identifiers would be quite useful, but we know of no such algorithm.

4.3.4 Link Protocols

Links, joining nodes in a network, are implemented with one or more *physical channels*, each delivering low-level messages called "packets". Packet delivery is not necessarily reliable. A *link protocol* is used to implement reliable message communication using unreliable physical channels. Of course, it is necessary to make some assumptions about the types of failures permitted for the physical channels. For example, channels might be assumed to lose and reorder messages, but not to duplicate or fabricate them. In addition, some liveness assumption on the physical channel is needed to ensure that messages are eventually delivered; a common assumption is that if infinitely many packets are sent, then eventually some message will be delivered.

The Alternating Bit Protocol is a link protocol that assumes the physical channel may lose packets but cannot reorder them. When a sender wishes to transmit a message, it assembles a packet consisting of the message and a single bit "header", and transmits this packet repeatedly on the physical channel. Upon receipt of the packet, the receiver sends the header bit back to the sender. When the sender receives a header bit that is the same as the one it is currently transmitting, it knows that its current message has been received and switches to the next message, using the opposite header bit.

This protocol does not work if the physical channels can reorder messages. In fact, Lynch, Mansour, and Fekete [LMF88] showed that no protocol with bounded headers can work over non-FIFO physical channels, if the best-case number of packets required to deliver each message must be bounded. Attiya et al. [AFWZ89] complete the picture by showing that this latter assumption is necessary—that there is a (not very practical) protocol using bounded headers if the best-case number of packets required to deliver one message is permitted to grow without bound.

Baratz and Siegel [BS88] developed link protocols that tolerate "crashes" of the participating nodes, with loss of information in the nodes' states. Their algorithm requires the node at each end of the link to have one bit of "stable memory" that survives crashes. It is shown in [LMF88] that this bit of stable memory is necessary.

Aho et al. [AUWY82] have studied the basic capabilities of finite-state link protocols.

4.4 Concurrency Control in Databases

A database consists of a collection of items that are individually read and written by the operations of programs called *transactions*. A *concurrency control algorithm* executes each transaction so it either acts like an atomic action, with no intervening steps of other transactions, or aborts and does nothing. This condition, called *serializability*, ensures that the system acts as if all transactions that are not aborted are executed in some serial order. This order must be consistent with the order in which any externally visible actions of the transactions occur. The serializability condition for databases is very similar to the atomic condition discussed earlier for shared variables.

By making transactions appear atomic, concurrency control makes the system easier to understand. For this reason, atomic transactions have been proposed as a basic construct in distributed programming languages and systems such as Argus [Lis85] and Camelot [STP*87].

Allowing transactions to be aborted permits more efficient concurrency control algorithms. An algorithm can make scheduling decisions that lead to faster execution but may produce a nonserializable execution; it aborts any transaction whose execution would not appear atomic. It is sometimes useful to abort a transaction for reasons other than maintaining serializability. The transaction might be running very slowly and holding needed resources, or the person who submitted the transaction could change his mind and want it aborted.

The simplest concurrency control algorithm is one that actually runs the transactions serially, one at a time. However, such an algorithm is not satisfactory because it eliminates the possibility of concurrent execution of transactions, even if they access disjoint sets of data items.

4.4.1 Techniques

Hundreds of papers have been written about concurrency control algorithms, and many techniques have been proposed. We discuss only the two most popular ones: locking and timestamps. We refer the reader to the textbook by Bernstein, Hadzilacos, and Goodman [BHG87] for a more complete survey of concurrency control algorithms and an exposition of some of the underlying theory, and to Lynch et al. [LMWF88, LMWF90] for a general theory of concurrency control algorithms.

Locking The concurrency control method used most often in commercial systems is *locking*. A locking algorithm requires a transaction to obtain a *lock* on each data item before accessing it, preventing conflicting operations on the item by different transactions. There are usually two kinds of locks: exclusive locks that enable the owner to read or write the item, and shared locks that enable the owner only to read it. Several transactions can hold shared locks on the same item, but a transaction cannot have an exclusive lock while any other transaction holds either kind of lock on that item.

In a classic paper, Eswaran et al. [KPET76] showed that serializability is guaranteed by *two-phase* locking, in which a transaction does not acquire any new locks after releasing a lock—for example, if it requests all locks at the beginning and releases them all at the end. However, if locks are acquired one at a time, deadlock is possible in which each member of some set of transactions is waiting for a lock held by another member of the set. Such deadlock must be detected, and the deadlock broken by aborting one or more waiting transactions. The effects of any aborted transaction must be undone; this may require saving the original values of all data items that have been modified by transactions which have not yet completed.

The notions of shared and exclusive locks can be generalized to other kinds of locks, depending on the semantics of the operations on the database—in particular, on which operations commute. These other classes of locks lead to more general and efficient concurrency control mechanisms than ones based only on shared and exclusive locks.

Timestamps Instead of using locks, some algorithms use timestamps (described in Section 2.3) to control access to data items. A timestamp is assigned to each transaction, and the algorithm ensures that transactions not aborted are executed as if they were run serially in the order specified by their timestamps. This serial execution order is obtained if operations to the same item are performed in timestamp order, where the timestamp of an operation is defined to be that of the transaction to which it belongs. One way to implement this condition is not to execute an operation on an item until all operations to that item with smaller timestamps have been executed. In a distributed database, waiting until no operation with an earlier timestamp can arrive may be expensive. Alternatively, one can abort a transaction if it tries to access a data item that has already been accessed by a transaction with a later timestamp.

So far, we have assumed that only a single version of the item is maintained. Additional flexibility can be achieved by keeping several earlier versions as well, since it is no longer necessary to abort a transaction when it accesses an item that has already been accessed by transactions with later timestamps. For example, if the transaction is just reading the item, the serial order can be preserved by reading an earlier version. Some of these earlier versions might be needed anyway to restore the item's value if a transaction is aborted.

While timestamps seem to offer some advantages over locking, almost all existing database systems use locking. This may be at least partly due to timestamp algorithms being more complicated than the commonly used locking methods.

4.4.2 Distribution Issues

In distributed systems, items can be located at multiple sites. A concurrency control algorithm must guarantee that all sites affected by a transaction agree on whether or not it is aborted. This agreement is obtained by a commit protocol (Section 4.2.4).

Copies of an item may be kept at several sites, to increase its availability in the event of site failure or to make reading the item more efficient. For transactions to appear atomic, they must provide the appearance of accessing a single copy. One method of ensuring this is to require each operation to be performed on some subset of the copies—a read using the most recent value from among the copies it reads. Atomicity is guaranteed if the transactions are serialized and the sets of copies of any item accessed by any two operations have at least one element in common—for example, if each read reads two copies and each write writes all but one copy.

4.4.3 Nested Transactions

The concept of a transaction has been generalized to *nested* transactions, which are transactions that can invoke subtransactions as well as execute operations on items. The nesting is described as a tree; each transaction is the parent of the subtransactions it invokes, and an added root transaction serves as the parent of all top-level transactions. Serializability is generalized to the requirement that for every node in the transaction tree, all its children appear to run serially. Algorithms based upon locking and timestamps have been devised for implementing this more general condition.

With nested transactions, failures can be handled by aborting a subtransaction without aborting its parent. The parent is informed that its child has aborted and can take corrective action. Nested transactions appear as a fundamental concept in the Argus distributed programming language [Lis85] and in the Camelot system [STP*87].

The framework presented in [LMWF88] is general enough for modeling nested transactions as well as ordinary single-level transactions.

References

[AAG87] Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, pages 358–370, Los Angeles, California, October 1987.

[AAHK86] Karl Abrahamson, Andrew Adler, Lisa Higham, and David Kirkpartrick. Probabilistic solitude verification on a ring. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 161–173, Calgary, Canada, August 1986.

[AFdR80] Krzysztof R. Apt, Nissim Francez, and Willem P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, July 1980.

[AFL83] Eshart Arjomandi, Michael J. Fischer, and Nancy A. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *Journal of the ACM*, 30(3):449–456, July 1983.

[AFWZ89] H. Attiya, M. Fischer, D. Wang, and L. Zuck. Reliable communication over an unreliable channel. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, Association for Computing Machinery, 1989.

[AGPV88] B. Awerbuch, O. Goldreich, D. Peleg, and R. Vainish. *On the Message Complexity of Broadcast: Basic Lower Bound*. Technical Memo TM-365, MIT Laboratory for Computer Science, July 1988.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[ASW88] Hagit Attiya, Marc Snir, and Manfred K. Warmuth. Computing on an anonymous ring. *Journal of the ACM*, 35(4):845–875, October 1988.

[AUWY82] A. V. Aho, J. Ullman, A. Wyner, and M. Yannakakis. Bounds on the size and tranmission rate of commumication protocols. *Computers and Mathematics with Applications*, 8(3):205–214, 1982.

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[Ben83] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 27–30, Association for Computing Machinery, New York, August 1983.

[BHG87] Philip A. Bernstein, V. Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1987.

[BJL*82] James E. Burns, Paul Jackson, Nancy A. Lynch, Michael J. Fischer, and Gary L. Peterson. Data requirements for implementation of n-process mutual exclusion using a single shared variable. *Journal of the ACM*, 29(1):183–205, January 1982.

[Blo88] B. Bloom. Constructing two-writer atomic registers. *IEEE Transactions On Computers*, 37(12):1506–1514, 1988.

[BP87] James E. Burns and Gary L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 222–231, Vancouver, British Columbia, Canada, August 1987.

[Bra85] Gabriel Bracha. An $O(\log n)$ expected rounds randomized byzantine generals algorithm. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 316–326, IEEE, May 1985.

[BS88] A. E. Baratz and A. Segall. Reliable link initialization prodecures. *IEEE Transactions on Communications*, COM-36(2):144–152, February 1988.

[BT87] G. Bracha and S. Toueg. Distributed deadlock detection. *Distributed Computing*, 2(3):127–138, 1987.

[CC84] B. A. Coan and B. Chor. A simple and efficient randomized byzantine agreement algorithm. In *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, pages 98–106, IEEE, 1984.

[CHEP71] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5(6):511–523, December 1971.

[CHM83] K. Mani Chandy, Laura M. Haas, and Jayadev Misra. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

[CHP71] P. J. Courtois, F. Heymans, and David L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, October 1971.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of a distributed system. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CM84] Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design*. Addison-Wesley, Reading, Massachusetts, 1988.

[DH79] Whitfield Diffie and Martin E. Hellman. Privacy and authentication: an introduction to cryptography. *Proceedings of the IEEE*, 67(3):397–427, March 1979.

[DHS84] Danny Dolev, Joe Halpern, and H. Raymond Strong. On the possibility and impossibility of achieving clock synchronization. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, pages 504–511, Association for Computing Machinery, Washington, D.C., 1984.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[Dij68] E. W. Dijkstra. The structure of the “the” multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[Dij71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[DM86] Cynthia Dwork and Yoram Moses. Knowledge and common knowledge in a Byzantine environment i: crash failures. In J. Halpern, editor, *Theoretical Aspects of Reasoning About Knowledge, Proceedings of the 1986 Conference*, pages 149–170, Morgan-Kaufmann, March 1986. Extended Abstract.

[Dol82] Danny Dolev. The Byzantine generals strike again. *Journal of Alorithms*, 3(1):14–30, March 1982.

[DS80] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[Eme] E. Allen Emerson. Temporal and modal logic. This Handbook.

[Fis83] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In Marek Karpinski, editor, *Foundations of Computation Theory*, pages 127–140, Springer-Verlag, 1983.

[FL81] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, June 1981.

[FL87] Greg N. Frederickson and Nancy A. Lynch. Electing a leader in a synchronous ring. *Journal of the ACM*, 34(1):98–115, January 1987.

[FLM86] M. J. Fischer, N. A. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, January 1986.

[FLP85] Michael J. Fischer, Nancy Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FM88] Paul Feldman and Silvio Micali. Optimal algorithms for byzantine agreement. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 148–161, Chicago, Illinois, May 1988.

[GC88] Eli Gafni and Ching-Tsun Chou. Understanding and verifying distributed algorithms using stratified decomposition. In *Proceedings of the 7th annual ACM Symposium on Principles of Distributed Computing*, pages 44–65, Toronto, Ontario, Canada, August 1988.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983.

[Gra78] J. N. Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, pages 393–481, Springer-Verlag, Berlin, Heidelberg, New York, 1978.

[Her88] Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *In Proceedings of the 7th annual ACM Symposium on Principles of Distributed Computing*, pages 276–290, Toronto, Ontario, Canada, August 1988.

[HM84] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pages 50–61, Association for Computing Machinery, New York, 1984.

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hum83] Pierre A. Humblet. A distributed algorithm for minimum weight directed spanning trees. *IEEE Transactions on Communications*, COM-31(6):756–762, June 1983.

[KPET76] R. A. Lorie K. P. Eswaran, J. N. Gray and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[Lam74] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam84] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, 6(2):254–280, April 1984.

[Lam86] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–101, 1986.

[Lam89] Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[LG81] G. M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.

[Lis85] Barbara Liskov. The argus language and system. In M. Paul and H. J. Siegert, editors, *Distributed Systems: Methods and Tools for Specification*, chapter 7, pages 343–430, Springer-Verlag, 1985.

[LMF88] Nancy A. Lynch, Yishay Mansour, and Alan Fekete. The data link layer: two impossibility results. In *In Proceedings of the 7th ACM Symposium on Principles of Distributed Computation*, pages 149–170, ACM SIGACT and ACM SIGOPS, Toronto, Canada, August 1988. Also, MIT Technical Memo, MIT/LCS/TM-355, May 1988.

[LMWF88] Nancy A. Lynch, Michael Merritt, William Weihl, and Alan Fekete. A theory of atomic transactions. In M. Gyssens, J. Paredaens, and D. Van Gucht, editors, *Second International Conference on Database Theory*, pages 41–71, Springer-Verlag, Bruges, Belgium, September 1988.

[LMWF90] Nancy A. Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan-Kaufmann, 1990(?). In preparation.

[LS84] S. S. Lam and A. U. Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10(4):325–342, July 1984.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[LT87] Nancy Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Symposium on the Principles of Distributed Computing*, pages 137–151, ACM, August 1987.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, Heidelberg, New York, 1980.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6(4):319–340, 1976.

[OL82] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[Pet82] Gary L. Peterson. An $O(n \log n)$ unidirectional algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems*, 4(4):758–762, October 1982.

[Pet83] Gary L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[PF77] G. Peterson and M. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pages 91–97, ACM, May 1977.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Symposium on the Foundations of Computer Science*, pages 46–57, ACM, November 1977.

[PSL80] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[Rab82] Michael O. Rabin. N-process mutual exclusion with bounded waiting by 4 log n-valued shared variable. *Journal of Computer and System Sciences*, 25(1):66–75, 1982.

[Ray86] M. Raynal. *Algorithms for Mutual Exclusion*. MIT Press, Cambridge, Massachusetts, 1986.

[RL81] Michael Rabin and Daniel Lehmann. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 133–138, ACM, Williamsburg, Virginia, January 1981.

[SA86] F. B. Schneider and G. R. Andrews. Concepts for concurrent programming. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, pages 669–716, Springer-Verlag, Berlin, Heidelberg, New York, 1986.

[Ske82] Marion Dale Skeen. *Crash Recovery in a Distributed Database System*. PhD thesis, University of California, Berkeley, May 1982.

[SM82] Richard L. Schwartz and P. Michael Melliar-Smith. From state machines to temporal logic: specification methods for protocol standards. *IEEE Transactions on Communications*, COM-30(12):2486–2496, December 1982.

[STP*87] Alfred Z. Spector, Dean Thompson, Randy F. Pausch, Jeffrey L. Eppinger, Dan Duchamp, Richard Draves, Dean S. Daniels, and Joshua J. Bloach. *Camelot: A Distributed Transaction Facility for Mach and the Internet—An Interim Report*. Technical Report CMU-CS-87-129, Carnegie Mellon University, June 17 1987.

[Thi85] P. S. Thiagarajan. Some aspects of net theory. In B. T. Denvir, W. T. Harwood, M. I. Jackson, and M. J. Wray, editors, *The Analysis of Concurrent Systems*, pages 26–54, Springer-Verlag, Berlin, Heidelberg, New York, 1985.

[Vit84] Paul Vitányi. Distributed elections in an archimedean ring of processors. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, pages 542–547, Association for Computing Machinery, New York, April 1984.

[WLL88] Jennifer Welch, Leslie Lamport, and Nancy Lynch. A lattice-structured proof of a minimum spanning tree algorithm. In Danny Dolev, editor, *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 28–43, Association for Computing Machinery, Inc., New York, August 1988.