In mechanics we used only one clock. But this was not very convenient, because we had to take all measurements in the vicinity of this one clock. Looking at the clock from a distance, ... we have always to remember that what we see now really happened earlier, just as we receive light from the sun eight minutes after it was emitted. We should have to make corrections, according to our distance from the clock, in all our time readings.

—Einstein and Infeld (1938)

Now what does that mean? If we look at the situation carefully we see that events that occur at two separated places at the same time, as seen by [an observer] in [a coordinate system], do not happen at the same time as viewed by [another observer] in [another coordinate system]. ... This circumstance is called “failure of simultaneity at a distance.” ...

—Feynman, Leighton, and Sands (1963)

INTRODUCTION

Let $G$ be a connected undirected graph with $n$ nodes and $m$ edges. Each node in $G$ stands for a sequential processing element and each edge represents a bidirectional communication channel that allows for the point-to-point exchange of messages between its end nodes. $G$ is then representative of a multitude of distributed-memory systems, such as computer networks and, more abstractly, systems of multiple agents that do not share memory. In this chapter, we understand a distributed algorithm to be any algorithm for execution by the nodes of $G$ as concurrent sequential threads that communicate with one another by passing messages over the edges of $G$. We let nodes be numbered $1, ..., n$.

The field of distributed algorithms is an outgrowth of the research on concurrent programming that some decades ago was an essential part of the research on operating systems. The field evolved to encompass both the message-passing computations that are our subject matter in this chapter and computations on shared memory. Through the years, key representative problems have been identified, and a core collection of fundamental techniques has been put together for use in the design and analysis of the algorithms. In a space-constrained chapter such as this one, it is unavoidable that some topics be left out (such as the entire sub field of shared-memory computations) while others are only skimmed over (such as computations in the presence of faults, which we cover later only briefly). However, we are at present fortunate to have several texts on distributed algorithms available for in-depth study. Of these, we single out the books by Barbosa (1996), Lynch (1996), Peleg (2000), Tel (2000), and Attiya and Welch (2004). Each favors a different approach to the subject and has its particular choice of topics. Combined, they offer a comprehensive body of knowledge on the field.

Most of the research on distributed algorithms has had a strongly theoretical flavor and has striven for the rigorous establishment of correctness—and performance-related properties. As the reader will notice throughout the chapter, often the approach has been to concentrate efforts on a few paradigmatic problems whose definitions aim at capturing the practical issues pertaining to real-world problems. For readers of a handbook such as this, it is important to realize that this approach has been very successful and that many practical solutions exist that implement what has been discovered. Two important examples are the applications of leader-election algorithms, particularly within the asynchronous transfer mode (ATM) layers that lie at the Internet’s backbone (Huang and McKinley 2000), and the use of algorithms to discover minimum-weight spanning trees on overlay networks for end-system multicast (Chu et al. 2002). Even conceptual problems whose treatment is fraught with difficulties, such as reaching agreement in faulty environments, have influenced real-world decisions: the four-way handshake that implements the closing of TCP connections (Kurose and Ross 2005),...
for example, is an attempt to handle the difficulties of such an agreement.

The two quotations that open this chapter are meant to stress what we deem to be the most fundamental issue underlying the study of distributed algorithms—that is, the issue of how to represent time, how to account for its passage, and how to refer to global entities without breaching any causality principles. We deal with this issue in the section on “Models of Computation,” albeit still somewhat superficially, with the introduction of the so-called synchronous and asynchronous models of computation that we use throughout the chapter. The section also contains representative distributed algorithms for each model. The asynchronous algorithms, in particular, are expanded and generalized in the section on “The Quintessential Asynchronous Algorithm” to yield a template for any asynchronous algorithm. We then continue in the section on “An Asynchronous Building Block” with the introduction of an important building block to be used in the construction of asynchronous algorithms. This building block is intended for the asynchronous establishment on $G$ of a rooted tree having certain desired properties.

In the section on “Events and Global States,” we return to time-related modeling issues and expand on the asynchronous model by introducing details that eventually lead to the notion of a consistent global state of an asynchronous computation. This is followed by a section on “Other Asynchronous Building Blocks,” in which two additional building blocks for the construction of asynchronous algorithms are discussed, namely the election of a leader and the distributed recording of a global state. Provisioned now with three important building blocks, we move in the next section to the description of an asynchronous algorithm that can be built out of some of them.

The issue of how to assess the computational complexity of a distributed algorithm is treated in the section on “Computational Complexity,” in which we discuss both time- and communication-related measures. With these in hand, we are equipped not only to analyze the worst-case performance of the algorithms discussed up to that section, but mainly we can then return to the interplay between synchronism and asynchronism and describe how to safely translate a synchronous algorithm into an asynchronous one while being able to assess the unavoidable change in computational complexity. We do this in the next section, in which we discuss the so-called synchronizers.

The section titled “Additional Topics” is devoted to brief discussions of further topics of interest within the field both from a theoretical perspective (viz. lower bounds and impossibility results, computing on anonymous systems, topological awareness, self-stabilization, consensus problems, correctness proofs, and the emergence of an all-encompassing theory of distributed computing) and from an applications-related one (resource sharing and the design of asynchronous algorithms for wireless networks and for large unstructured networks of generally unknown topologies—the so-called complex networks). To finalize, we give conclusions and closing remarks.

MODELS OF COMPUTATION

We give each model of distributed computation by specifying how time elapses at each node of $G$, how the elapsing of time at one node relates to what happens at another node, and how the passing of messages relates to time. For most of the chapter we assume that nodes and edges never fail.

The Synchronous Model

The synchronous model of distributed computation is characterized by the existence of a single clock that makes all nodes work in lockstep at the occurrence of its pulses, numbered by the integer $s \geq 0$. The model also assumes that the local computation occurring at each node for each pulse takes no time, and also that messages sent between neighbors in $G$ as part of the local computation for pulse $s$ are delivered before the occurrence of pulse $s + 1$. The assumptions of the synchronous model are unrealistic, but allow for great simplifications in the design of distributed algorithms, as we exemplify next.

Example: breadth-first numbering

This problem asks that each node $i$ in $G$ be labeled with the integer $l_i$ such that the number of edges of on the shortest path (i.e., the distance) from node $i$ to node 1 is $l_i$.

A synchronous algorithm to do this is initiated by node 1, which at pulse $s = 0$ lets $l_1 = 0$ and sends a message to all its neighbors. For $s > 0$, any node $i$ receiving a message for the first time lets $l_i = s$ and also sends a message to all its neighbors. Clearly, the problem can be solved within $\text{diam}(G) + 1$ pulses, where $\text{diam}(G)$ is the diameter (greatest distance between two nodes) of $G$. The number of messages used is $\sum d_i = 2m$, where $d_i$ is the degree of node $i$.

Example: leader election in a complete graph

When node $i$ has an identification $id_i$ that is distinct from all other nodes' and all identifications are totally ordered by $\prec$, this problem asks that all nodes in $G$ determine the node $k$ for which $id_k = \min \{id_i\}$. A nontrivial synchronous algorithm to do this on a complete graph is the following (Afek and Gafni 1991). Initially all nodes are candidates. At each even pulse, a node $i$ that remains a candidate sends $id_i$ to a new group of neighbors. At each odd pulse, every node compares all the identifications sent to it in the previous pulse to the least identification it has ever seen and, correspondingly, sends out at most one positive reply (to the owner of the new least identification, if any). A node remains a candidate if it receives positive replies to all identifications it sent out. If at pulse $s = 2z$, $z = 0, 1, \ldots$, the group of neighbors to which a candidate's identification is sent has size $2^z$, then clearly the number of pulses required to elect a leader is $O(\log n)$. As for the number of messages, notice that the number of candidates remaining at pulse $s = 2z$ for $z > 0$ is the number of nodes that were sent $2^{-z}$ positive replies at pulse $s - 1$—that is, at most $n/2^{z-1}$. The overall number of messages includes the initial $n$ identifications, the $(n/2^{z-1})2^{z}$ identifications sent for each $z > 0$, and finally the $O(n \log n)$ positive replies. These add up to $O(n \log n)$ messages.

Both examples take explicit advantage of the assumptions of the synchronous model. In the case of breadth-first numbering, this is revealed in the one-to-one correspondence between a node's distance to node 1 and the earliest pulse at which it can be reached by a message initially propagated by node 1. In the case of leader election
when $G$ is a complete graph, it is revealed in a subtler manner by the tacit information that, in the synchronous model, necessarily accompanies the absence of a message. The algorithm in the example exploits this by letting each node group all identifications sent to it before deciding whether to send out a positive reply, and also by dispensing with the need for negative replies.

**The Asynchronous Model**

In the asynchronous model, each node has its own local, independent clock, and computes, possibly sending out messages to its neighbors, upon the receipt of each message. Message delivery, even though guaranteed to occur, is in no way correlated to either the clock of the sender or of the receiver. The reactive response to an arriving message is always atomic—that is, it is never interrupted by the arrival of new messages, which are then queued up for future treatment. These assumptions, clearly, call for the existence of at least one initiator, that is, at least one node that performs some initial local computation, and sends out messages, without the need for an arriving message to respond to. We henceforth consider asynchronous algorithm nearly exclusively. They are in general more complex than synchronous algorithms; we start with very simple examples (Segall 1983).

**Example: flooding** Let $I$ be a piece of information to be disseminated through the nodes of $G$ from a single initiator, say node 1. In the absence of any further structure on $G$, one solution is to flood the graph: node 1 sends $I$ to all its neighbors; all other nodes, upon receiving $I$ for the first time, do the same. Clearly, this requires $2m$ copies of $I$ to be sent out.

**Example: flooding with feedback** Suppose that, in addition to disseminating $I$ as in the previous example, we also want node 1 to be informed that all other nodes have already been reached by the dissemination when this happens. Let $p_i$ be a variable local to node $i$. We use flooding with feedback: node 1 sends $I$ to all its neighbors; node $i \neq 1$, upon receiving $I$ for the first time, say from node $j$, lets $p_i = j$ and sends $I$ on to all its neighbors but $j$; node $j$ is sent $I$ back only when node $i$ has received $I$ from all its neighbors. The number of copies of $I$ required is still $2m$, but now a simple inductive argument on the spanning tree rooted at node 1 that the $p_i$ variables establish on $G$ shows that node 1, upon having received $I$ from all its neighbors, is certain that the dissemination has finished (Figure 1(a)).

The first example is valid for any number of concurrent initiators. The second one, with its inherent ability to anchor the entire asynchronous computation, via the rooted tree, at the single initiator, gives rise to both a generic template for gracefully terminating asynchronous algorithms (cf. the section on “The Quintessential Asynchronous Algorithm”) and also a fundamental building block for the development of asynchronous algorithms (cf. the section on “An Asynchronous Building Block”).

**THE QUINTESSENTIAL ASYNCHRONOUS ALGORITHM**

Any asynchronous algorithm with one single initiator can be cast into the template we give in this section. In the case of algorithms that rely on the flexibility of multiple concurrent initiators, the same property continues to be ultimately valid in its essence, since conceptually we can always add a new node to $G$, connect this node to the desired initiators, and then let it be the sole initiator instead. Our template is based on the elegant method laid down by Dijkstra and Scholten (1980) for detecting the termination of asynchronous computations that have one single initiator.

We start with a generic template for an asynchronous computation with one single initiator that does not require the initiator to be informed of termination. Such a template requires that the initial action to be performed by the initiator be specified, and also that for all nodes (including the initiator) a specification be given as to how to respond to the reception of each possible message type during the computation. As a shorthand, we refer to such messages generically as $msg$, always bearing in mind that $msg$ may stand for several different message types, each requiring a different treatment when received.

The generic template we seek in this section is an augmentation of the simple template we just outlined that aims at ensuring that the initiator be informed of termination when it occurs. It uses the same $p_i$ variables of the flooding with feedback algorithm for basically the same purpose. In addition, it is based on the premise that every $msg$ message is explicitly acknowledged—that is, for every $msg$ that node $i$ sends node $j$ there has to correspond an
acknowledgment message sent by node $j$ to node $i$. We let such a message be generically denoted by $ack$. Deciding when to send each $ack$ is the crux of the underlying method. For node $i$, a counter $c_i$, initially set to 0, is used to keep track of how many $ack$ messages node $i$ still expects to receive.

The following three actions specify the template asynchronous algorithm. As before, we let node 1 be the algorithm’s sole initiator.

1. **Spontaneous initiation at node 1**—Perform some local computation, possibly sending out $msg$ messages and incrementing $c_i$ by the number of messages sent.

2. **Upon the reception of $msg$ from node $j$ at node $i$**—Perform some local computation, possibly sending out $msg$ messages and incrementing $c_i$ by the number of messages sent. Then do one of the following:
   - (a) If $c_i = 0$ held when $msg$ was received and $c_i > 0$ holds now, then let $p_i = j$;
   - (b) Otherwise, send $ack$ to node $j$.

3. **Upon the reception of $ack$ at node $i$**—Decrement $c_i$ and check whether $c_i = 0$. In the affirmative case, conclude that termination has occurred if $i = 1$, or send $ack$ to $p_i$, if $i > 1$.

Any asynchronous computation conforming to this template spreads out from node 1 and, if termination ever occurs, eventually contracts back onto it. In the meantime, the computation remains anchored at node 1 through the tree rooted at node 1 that the variables $p_i$ induce if $c_i > 0$. This tree grows with the incorporation of node $i > 1$ into it when in action 2(a) node $i$ lets $p_i = j$; likewise, it contracts by shedding node $i > 1$ when in action 3 node $i$ sends $ack$ to $p_i$. As the computation progresses, such a node may enter and leave the tree several times, each time through a possibly different path from node 1 to it in $G$.

For computations that do terminate, eventually the tree stops growing and then the same inductive argument used to demonstrate the correctness of flooding with feedback can be used.

**AN ASYNCHRONOUS BUILDING BLOCK**

This recurring idea that building trees on $G$ asynchronously can be used to anchor asynchronous computations for proper termination signaling can be made more general and be applied to certain portions of the computation only. That is, it is possible to structure the tree-building procedure in such a way that it becomes a building block for the construction of more complex distributed algorithms. We specify such a building block now and use it to construct the algorithm in a later section as an example of its use.

Let $N_i$ denote the set of neighbors of node $i$ in $G$. Let $N_i^-$ be a subset of $N_i$. Also, let $S_i$ and $T_i$ be Boolean conditions relative to node $i$, used respectively to indicate, when they become true, that node $i$ is to be the sole initiator of the construction of a tree rooted at it, and that node $i$, not being the initiator, is to participate in such a construction as it is reached by its messages. The following three actions indicate how to grow and contract such a tree. We let node $k$ be the assumed initiator of the algorithm and root of the tree. We use the same $msg$ and $ack$ messages as in the section on “The Quintessential Asynchronous Algorithm.” Counter $c_i$ is also used at each node $i$ as in that section.

4. **Initiation at node $k$ when $S_k$ becomes true**—If $N_k^- \neq \emptyset$, then send $msg$ to all nodes in $N_k^-$ and increment $c_i$ by $|N_k^-|$.

5. **Upon the reception of $msg$ from node $j$ at node $i$**—If $i \neq k$ and receiving $msg$ has caused $T_i$ to become true, and also $N_i^- \neq \emptyset$, then send $msg$ to all nodes in $N_i^-$, increment $c_i$ by $|N_i^-|$, and let $p_i = j$. Otherwise, send $ack$ to node $j$.

6. **Upon the reception of $ack$ at node $i$**—Decrement $c_i$ and check whether $c_i = 0$. In the affirmative case, conclude that the tree has contracted back onto its root if $i = k$, or send $ack$ to $p_i$, if $i \neq k$.

Actions 4 to 6 promote the growth of a tree on $G$ that is rooted at node $k$. Its creation starts when $S_k$ becomes true and is joined by every node $i \neq k$ for which $T_i$ becomes true as $msg$ messages arrive and $N_i^-$ is nonempty. Its contraction is started concurrently at every node $i \neq k$ for which $T_i$ never becomes true or $N_i^- = \emptyset$, and also as nodes receive copies of $msg$ other than the first, and continues with the ensuing sending of the $ack$ messages. Obviously, letting $N_i^- = N_i$ for all $i$ and setting $T_i$ to become true for all $i > 1$ as the first $msg$ arrives yields the asynchronous algorithm for flooding with feedback. In this case, $msg = ack = 1$ and the flooding starts when $S_i$ becomes true.

**EVENTS AND GLOBAL STATES**

The design of an asynchronous algorithm must be resilient to any degree of the uncertainties that underlie the asynchronous model. Such uncertainties refer to the fact that local clocks are in no way correlated to one another, and also to the unpredictable delays that messages may undergo as they are transmitted on the edges of $G$. What this amounts to is that different executions of the same asynchronous algorithm may, from the local standpoint of a certain node, entail the reception of a different sequence of messages for each execution and therefore a different sequence of actions as well. Characterizing what is meant by an execution precisely is then crucial, as it affects directly the conception of certain asynchronous building blocks and also allows complexity measures to be defined correctly.

In what follows, and also in some of the forthcoming sections, we use a directed graph $D$ in lieu of the undirected graph $G$ we have been using. The nodes of $D$ are the same as those of $G$, and its edges, by contrast, indicate the possibility of unidirectional communication between the nodes. We do this substitution for the sake of generality, since an undirected graph with the bidirectional-channel interpretation we have ascribed to the edges of $G$ is equivalent to the particular case of $D$ that has two directed edges between every two nodes that are neighbors in $G$, one in each direction. We let $c$ denote the number of directed edges in $D$. 
Our definition of an execution of an asynchronous algorithm is based on the seminal work of Lampert (1978). For an asynchronous algorithm A, we define an execution of A to be a set $V_A$ of so-called events (Figure 1(b)). An event $v$ is a sextuple containing the following elements:

- **$node(v)$** is the node of $D$ at which $v$ happens;
- **$time(v)$** is the time, as given by the local clock of $node(v)$, right after the occurrence of $v$;
- **$in_msg(v)$** is the message, if any, whose reception at $node(v)$ triggers the occurrence of $v$;
- **$Out_msgs(v)$** is the set of the messages, if any, that $node(v)$ sends out as a consequence of the occurrence of $v$;
- **$p_state(v)$** is the local state of $node(v)$ that precedes the occurrence of $v$;
- **$s_state(v)$** is the local state of $node(v)$ that succeeds the occurrence of $v$, that is, at time $time(v)$.

Clearly, this definition of an event captures the essential nature of an asynchronous algorithm, allowing for the first actions of initiators and for the subsequent processing by all nodes as they react to the reception of individual messages.

The events in $V_A$ are obviously not independent of one another. In fact, the definition of the following binary relation, denoted by $B$, is quite natural. For $v, v' \in V_A$, we say that $(v, v') \in B$ if one of the two following possibilities holds: either $node(v) = node(v')$ and $time(v) < time(v')$ with no intervening events between them; or $node(v) \neq node(v')$ and $in_msg(v') \in Out_msgs(v)$. In other words, $B$ characterizes the situations in which $v'$ is an immediate predecessor of $v'$, either occurring in immediate succession at the same node, or occurring at different nodes, provided the message that triggers the occurrence of $v'$ is sent as a consequence of the occurrence of $v$.

This relation $B$ gives rise to two important entities. The first is an acyclic directed graph, called the precedence graph of execution $V_A$; its node set is $V_A$ and its edge set is $B$ (Figure 1(c)). The second important entity is the transitive closure $B^*$ of $B$, which is known as the happened-before relation; $B^*$ is a partial order on $V_A$ and allows the precise definition of the past of an event $v$ as $Past(v) = \{ v' \in V_A \mid (v', v) \in B^* \}$ and of $v$’s future as $Future(v) = \{ v' \in V_A \mid (v, v') \in B^* \}$. It also allows concurrent events in $V_A$ to be defined precisely: $v, v' \in V_A$ are concurrent if neither $(v, v') \in B^*$ nor $(v', v) \in B^*$.

The asynchronous model, by definition, precludes the precise treatment of global properties in time-related terms. This, of course, is because of the absence of a global clock, but the need remains for us to be able to refer to global properties. For example, the termination property that is detected as, in the algorithm of the section on “The Quintessential Asynchronous Algorithm,” node 1 sets $c_1$ to 0 is a global property that makes sense even if we eliminate all $ack$ messages from that algorithm: all this elimination achieves is that node 1 is no longer capable of detecting termination, but the notion of termination remains valid, and it is legitimate for us to want to refer to it globally. The way to achieve this is by defining precisely what a global state of the execution $V_A$ is. We present this definition in two stages.

In the first stage, we first partition $V_A$ into the sets $V_1(1),...,V_1(n)$, where $V_1(i)$ is the set of events $v$ for which $node(v) = i$. Then we further partition each $V_1(i)$ into $V_2(i)$ and $V_3(i)$ in such a way that $time(v) < time(v')$ for any $v \in V_2(i), v' \in V_3(i)$. If we now regroup these sets into $V_3 = \cup V_3(i)$ and $V_2' = \cup V_2(i)$, then the new partition of $V_A$ into $V_1, V_2,$ and $V_3$ can be seen to define a global state of the system represented by the directed graph $D$. In this global state, the local state of node $i$ is either its initial state (if $V_3(i) = \emptyset$) or $s_state(v)$, where $v$ is the event in $V_2(i)$ for which $time(v)$ is greatest. The global state also includes messages in transit on the edges of $D$; these are the messages sent as a consequence of the occurrence of $v$ such that $v' \in V_1$ and $in_msg(v') \in Out_msgs(v)$ for some $v \in V_3$.

Global states defined in this way can be easily seen to give rise to causal inconsistencies with respect to the partial order $B^*$. In the definition's second stage, we then complete it by requiring that $Past(v) \subseteq V_1$ for all $v \in V_1$ (an equivalent statement of this requirement is that $Future(v) \subseteq V_3$ for all $v \in V_3$). We sometimes refer to $V_1$ as the global state's past and to $V_3$ as its future. Global states are illustrated in Figure 1(d).

**Vector Clocks and Beyond**

This theory has been extended by the incorporation of several other time-related notions for asynchronous algorithms. These extensions can be found in the articles collected by Yang and Marsland (1994) and also elsewhere (e.g., Drummond and Barbosa 2003). One of the strongest motivations has been the design of algorithms for detecting various types of global predicates, such as the ones of Drummond and Barbosa (1996). We close this section by briefly discussing the notion of a vector clock. The vector clock of node $i$ at time $time(v)$, with $v$ such that $node(v) = i$, is the size-$n$ one-dimensional array $K_i$ whose $j$th component is

$$K_i[j] = \begin{cases} time(v), & \text{if } j = i; \\ time(pred(v)), & \text{if } j \neq i, \end{cases}$$

where $pred(v)$ is the latest predecessor of $v$ at node $j$, that is, the event $v' \in V_j(i)$ such that $(v', v) \in B^*$ for which $time(v')$ is greatest (if no such event exists, then we assume $time(pred(v)) = 0$). It is easy to see that there exists a global state of $V_j$ in which $K_i$ is the node-state part. Also, if every node attaches its current vector clock to all messages it sends, and furthermore updates its vector clock, whenever it receives a message, to the component-wise maximum of the vector clock that comes in the message and its own, then the result remains a valid vector clock. The extension of vector clocks to higher-dimensional matrix clocks follows similar principles and allows for consistent local representations of arbitrary-depth portions of an event’s past in execution $V_A$.

**OTHER ASYNCHRONOUS BUILDING BLOCKS**

In addition to the tree-constructing asynchronous building block of the section on “An Asynchronous Building Block,” two others are generally regarded as being just as fundamental. We discuss them next.
Leader Election

In the section on "Models of Computation," we introduced the leader-election problem as the problem of finding the node whose identification is the minimum. The problem is well posed, because we assumed that all nodes have distinct identifications totally ordered by $<$ and that all nodes are candidates initially. In this section we consider a more general definition of the problem: not only is $G$ no longer assumed to be a complete graph, but also not all nodes are candidates initially. In addition, the problem does not ask for the candidate of least identification, but asks instead that all nodes agree on some candidate’s identification.

When posed like this, the problem is very closely related to the problem of finding a minimum-weight spanning tree on $G$ when edge weights are all distinct and totally ordered by $<$ (and hence $G$ has a unique minimum-weight spanning tree). The reason that the two problems are so close to each other is that, in several solutions of the latter problem (Gallager, Humblet, and Spira 1983; Chin and Ting 1990), when the minimum-weight spanning tree is finally obtained, we automatically have a core edge whose end nodes, having distinct identifications, can decide between which one becomes leader if at least one of them is a candidate. If neither is a candidate, then they can take upon themselves the task of deciding which of the candidates is to become leader. Obtaining their identifications and disseminating through all nodes the leader’s identification is achieved rather simply by computing exclusively on the edges of the tree. The entire procedure, including finding the tree, can be designed so that the number of messages it requires is $O(m + n \log n)$.

It then remains for appropriate weights to be set for all edges of $G$. Any set of distinct weights ordered by $<$ will do, including the adoption of the ordered pair $(\min(id_i, id_j), \max(id_i, id_j))$ as the weight of the edge between nodes $i$ and $j$. In this case, weights are ordered lexicographically by $<$, and this is how comparisons are to be made as the minimum-weight spanning tree is built.

Global-State Recording

Consider an arbitrary execution $V_s$ of an asynchronous algorithm $A$. Let $X^i_1, X^i_2$ and $Y^i_1, Y^i_2$ be distinct partitions of $V_s$ that define global states as explained in the section on "Events and Global States." We say that the latter of these global states is in the future of the former one if $X^i_1 \subset Y^i_2$ (equivalently, if $Y^i_1 \subset X^i_2$). Now let $P$ be a global predicate on execution $V_s$. We say that $P$ is stable if $P$ is true of all global states that are in the future of a global state of which $P$ is true. One example of stable predicates is termination, which is true of a global state if and only if all nodes are idle and all edges are empty in that global state.

One possible approach to the detection of stable predicates is to repeatedly record global states of $V_s$ until one is found of which the predicate is true. The ability to make such recordings is then crucial, and we now review an asynchronous algorithm to do it. As in the section on "Events and Global States," for the remainder of this section we revert to using the directed graph $D$ in lieu of the undirected $G$.

The algorithm we give is the one of Chandy and Lamport (1985). Like all algorithms for recording global states, it is to be viewed as a separate computation that is superimposed on the computation whose execution is $V_s$ (this execution is then referred to as a substrate). Even though $A$ and the recording algorithm are separate entities, it is the interaction between them that allows for the recording of global states of $V_s$. So the two algorithms share the nodes of $D$ (whose processing capabilities are split between the two as their actions are executed atomically) and also its edges.

Similarly to our earlier practice, we refer to each message of the substrate generically as msg. The recording algorithm uses marker messages exclusively, even though it also handles, without interpreting them, the msg messages. During recording, each node is responsible for recording its own local state and also the states, in the form of sequences of messages, of all edges that in $D$ are directed toward it. We use $M_{s}(j)$ to denote the sequence, initially empty, into which node $i$ records messages it receives from $j$ if $(j, i)$ is a directed edge of $D$.

The recording admits any number of concurrent initiators. An initiator records its own local state and sends marker on all its outgoing edges. Any other node, upon receiving the first marker, does the same. From then on, node $i$ (initiator or otherwise) monitors the msg messages it receives: if $(j, i)$ is an edge of $D$, then every msg received from node $j$ before a marker is received from that same node is appended to $M_{s}(j)$.

This algorithm, evidently, is nothing but the obvious extension to directed graphs of the flooding algorithm of the section on "Models of Computation," naturally with the necessary provisions for node- and edge-state recording. If $D$ is strongly connected, then assuredly all nodes participate in the recording and, overall, exactly $c$ marker’s are sent. It is also possible to prove that, if in addition $D$'s edges deliver messages in the first-in, first-out (FIFO) order, then all the information the algorithm records does indeed constitute a global state. In this global state, all (and only) messages recorded in $M_{s}(j)$ have the following property: they were sent before node $j$’s recording of its own local state and received after node $i$’s. This property is the key to extending the algorithm to the case of non-FIFO message delivery.

AN EXAMPLE

We draw this section's example from the area of distributed computations for resource sharing. In this case, the nodes of $G$ compete with one another for the use of shared resources, and we assume that the sharing policy they follow defines a dynamic subgraph of $G$ that reflects, along the computation’s global states, the wait of nodes for one another. Even though $G$ is an undirected graph, this subgraph has directed edges: if the directed edge $(i, j)$ is part of it in a certain global state, then in that global state node $i$ is blocked and waiting for a signal from node $j$ to help release it. We denote this subgraph of $G$ by $W$. By the nature of $W$, its sinks (if any) are the only unblocked nodes, and the possibility of deadlocks is a reality.

A rich set of graph-theoretic concepts exists for the treatment of deadlocks in $W$ (Barbosa 2002). In particular,
since the existence of a deadlock in $W$ is a stable property, recording a global state of the resource-sharing computation and checking the corresponding (static) $W$ for the occurrence of some sufficient condition for deadlocks to occur is a common approach (we refer the reader to the example given by Misra and Chandy (1982), which is a clever, albeit concise, application of the template of the section on "The Quintessential Asynchronous Algorithm").

One example that illustrates the use of the tree-construction building block of the section on "An Asynchronous Building Block" is the following (Bracha and Toueg 1987). Suppose that the resource-sharing computation is such that node $i$ must receive at least $x_i$ signals from its out-neighbors in $W$ to be released. In order for node 1 to detect whether it is in deadlock according to a recorded $W$, it first builds a tree following actions 4 to 6 with: $k = 1$, $S_1$ becoming true when node 1 suspects it is in deadlock, $T_i$ becoming true when node $i \neq k$ receives the first message, and finally $N_i^i$, the set of out-neighbors of node $i$ in $W$. This tree unfolds out from node 1 through the nodes' out-neighbors. If any sinks are reached, then each one builds another tree concurrently with the other sinks, before allowing the original tree to contract. Each sink-rooted tree follows actions 4 to 6 with: $k$ the corresponding sink's number, $S_i$ becoming true when node $k$ is reached by the messages constructing the original tree, $T_i$ becoming true when (non-sink) node $i \neq k$ has just received $x_i$ of the messages used to expand the sink-rooted trees, and finally $N_i^i$, the set of in-neighbors of node $i$ in $W$. After the original tree contracts back on node 1, it decides for deadlock if and only if $T_i$ is false (note that $T_i$ is defined only for the sink-initiated tree constructions). This algorithm requires no more messages than twice the number of edges in $W$. An illustration is given in Figure 2.

**COMPUTATIONAL COMPLEXITY**

So far we have only superficially touched the issue of computational complexity. In the field of distributed computing, the complexity measures of interest are the time for completion of the algorithm and the overall number of messages used. In previous sections we have only given complete complexity figures for the two simple synchronous algorithms of the section on "Models of Computation"; for all asynchronous algorithms, we limited ourselves to discussing the number of messages used. We now give precise definitions in both cases; we concentrate on the case of computational complexity. In the field of distributed computing, the complexity measures of interest are the time for completion of the algorithm and the overall number of messages used. In what follows, we let $G$ denote the class of all graphs for which a certain algorithm is designed (e.g., all connected graphs, all rings, etc.).

For an asynchronous algorithm $A$ and a graph $G \in G$, let $V_\ell(G)$ be the set of all possible executions of $A$ on $G$. In the section on "Events and Global States," we emphasized the possibility of multiple distinct executions of $A$ as they stem from the inherent unpredictability of the asynchronous model, but clearly there can be variation due to differences in initial local conditions and also to probabilistic decisions. We denote the number of messages required for running $A$ on $G$ during execution $V_\ell$ by $\text{Messages}(A,G,V_\ell)$. Clearly, assessing this number correctly is a matter of counting message exchanges during the execution. The worst-case message complexity of algorithm $A$, denoted by $M\text{Compl}(A)$, is defined as

$$M\text{Compl}(A) = \max_{G \in G} \max_{V_\ell \in V_\ell(G)} \text{Messages}(A,G,V_\ell).$$

That is, for fixed $G$ we first find the number of messages exchanged during the execution of $A$ on $G$ that requires the most messages. Then we take the greatest of these numbers over $G$. Naturally, the resulting figure depends on the parameters associated with at least one $G \in G$, such as $n$, $m$, diam$(G)$, or possibly others; here, and henceforth, we refrain from indicating this dependency explicitly for notational ease. All message complexities we have heretofore in the chapter for asynchronous algorithms conform to this definition, as one may easily check.
A completely analogous definition can be given for the worst-case time complexity of \( A \), which we denote by \( T\text{Compl}(A) \):

\[
T\text{Compl}(A) = \max_{G} \max_{V_{i}} \text{Time}(A, G, V_{i}),
\]

where \( \text{Time}(A, G, V_{i}) \) is the time required for running \( A \) on \( G \) along execution \( V_{i} \). The absence of a global clock in the asynchronous model requires that we approach the definition of this latter quantity with care. For cases in which the guarantee of a single initiator exists, a possibility one might think of would be to let this time be as given by the initiator’s local clock. But this approach is avoided in general, because it makes the eventual time complexity depend on a specific clock and on the rate at which time progresses according to that clock.

A better approach is to recognize that the time expended by an asynchronous algorithm is determined primarily by the causal flow of messages on \( G \) during execution. For this reason, we assume local computation to take no time (just as in the synchronous model) and let \( \text{Time}(A, G, V_{i}) \) be given by the length of the longest causal chain in \( V_{i} \) involving the reception of a message and the sending of another as a consequence. More precisely, we take the precedence graph of execution \( V_{i} \) and assign weights to each of its edges: either 1 (if it is a message-related edge) or 0. Then we let \( \text{Time}(A, G, V_{i}) \) be the length of the longest weighted directed path on the precedence graph of \( V_{i} \).

As we look back on the asynchronous algorithms seen previously in the chapter, we see that the flooding-based ones (both the two of the section on “Models of Computation” and the algorithm for global-state recording) all have an \( O(n) \) worst-case time complexity. We also remark that it is possible to design an asynchronous algorithm for minimum-weight spanning trees that runs in \( O(n \log^* n) \) time in the worst case (\( \log^* \) denotes the number of successive applications of the log operator required to yield a number less than 1). As for the deadlock detection algorithm of the section above in which we give an example, it runs in \( O(w) \) time in the worst case, where \( w \) is the number of nodes of \( W \).

We finalize by pointing out that the case of synchronous algorithms is entirely analogous, provided we understand that the possibility of multiple executions has now nothing to do with the model of computation. Furthermore, the time required to run a synchronous algorithm \( S \) on \( G \) along a certain execution can be assessed by simply counting pulses. Henceforth, we let \( M\text{Compl}(S) \) and \( T\text{Compl}(S) \) denote, respectively, the worst-case message and time complexities of \( S \).

SYNCHRONIZERS

As we remarked in the section on “Models of Computation,” the synchronous model is unrealistic. The reason that it remains important despite this fact is that designing algorithms under its assumptions is often considerably simpler than designing for the asynchronous model. For example, the synchronous algorithm given in the previously mentioned section for numbering the nodes of \( G \) breadth-first is amazingly simple; an asynchronous algorithm, by comparison, would require considerable additional control and be significantly more complex.

But what really guarantees the place of the synchronous model in the study of distributed algorithms is the certainty that every synchronous algorithm can be correctly and automatically translated into an equivalent asynchronous one. This is achieved by transforming the operation of the synchronous algorithm at each pulse \( s \geq 0 \) into a fragment of an asynchronous algorithm. The desired asynchronous algorithm is obtained by piecing together these fragments for all values of \( s \). Such fragments are known as synchronizers (Awerbuch 1985).

Let \( \text{Sync} \) be a synchronizer and suppose its worst-case message and time complexities (the latter in the asynchronous sense) per pulse are \( M(\text{Sync}) \) and \( T(\text{Sync}) \), respectively. Suppose also that initializing \( \text{Sync} \) requires \( M_{d}(\text{Sync}) \) messages and \( T_{d}(\text{Sync}) \) time at most. From the general description we have given so far of a synchronizer, it is clear that the asynchronous algorithm \( A \) resulting from transforming a synchronous algorithm \( S \) via synchronizer \( \text{Sync} \) has worst-case message and time complexities given, respectively, by

\[
M\text{Compl}(A) = M\text{Compl}(S) + M_{d}(\text{Sync}) + T\text{Compl}(S)M(\text{Sync})
\]

and

\[
T\text{Compl}(A) = T_{d}(\text{Sync}) + T\text{Compl}(S)T(\text{Sync}).
\]

The essential property that needs to be guaranteed before a node executes its portion of the synchronous algorithm corresponding to pulse \( s + 1 \) for \( s \geq 0 \) is that all messages sent to it at pulse \( s \) have reached it. This can be taken for granted when the assumptions of the synchronous model are in effect, but lifting them requires special actions to ensure that the property continues to hold. Let us call a node safe with respect to pulse \( s \) when all messages it sends during that pulse have reached their destinations. Clearly, in order for a node to proceed to pulse \( s + 1 \) it suffices that all its neighbors be safe with respect to pulse \( s \). The task of a synchronizer is to convey this safety information to all nodes for each pulse. We start by having all messages of \( S \) explicitly acknowledged by an \( \text{ack} \) message; clearly, this has no effect on \( M\text{Compl}(S) \) or, provided no \( \text{ack} \) is ever withheld for later transmittal, on \( T\text{Compl}(S) \).

Basic Synchronizers

The first basic synchronizer is called \( a \) and conveys the safety information directly between neighbors. When a node has received all \( \text{ack} \) messages corresponding to the messages it sent in pulse \( s \) it sends a \( \text{safe(s)} \) message to all its neighbors. Upon receiving a \( \text{safe(s)} \) message from all its neighbors, a node may proceed to pulse \( s + 1 \). We clearly have \( M(a) = O(m) \) and \( T(a) = O(1) \). Initializing synchronizer \( a \) requires that \( G \) be flooded with a signal for all nodes to start pulse 0, and so we have \( M_{d}(a) = O(m) \) and \( T_{d}(a) = O(n) \). Another basic synchronizer, called \( b \), requires a rooted tree to have been built on \( G \) beforehand. The root floods the tree, with feedback, with successive signals for the nodes to execute the succession of pulses.
The feedback progresses past a node only when that node has received \( \text{ack} \) messages for all messages it sent during the current pulse. Clearly, \( M(\beta) = T(\beta) = O(n) \). As for \( M(\alpha) \) and \( T(\beta) \), they depend on the complexities of electing a leader (cf. the section on "Other Asynchronous Building Blocks").

A Parameterized Synchronizer

Synchronizers \( \alpha \) and \( \beta \) may be combined in a parameterized way to yield another synchronizer. This combination requires the establishment on \( G \) of a rooted forest with a number \( k \) of trees and of preferential edges between neighboring trees. Synchronization takes place at two levels: inside each tree it works as synchronizer \( \beta \); at a higher level, at which we consider a graph whose nodes are the trees’ roots and whose edges join roots between which a simple path exists in \( G \) containing only tree edges and one preferential edge, it works as synchronizer \( \alpha \). If the largest tree has \( O(f(G)) \) edges, the number of preferential edges is \( O(g(G)) \), and moreover the greatest tree height is \( O(h(G)) \), then we have, letting \( \gamma \) denote this hybrid synchronizer, \( M(\gamma) = O(k f(G) + g(G)) \) and \( T(\gamma) = O(h(G)) \). The synchronizer \( \gamma \) of Awerbuch (1985) has \( M(\gamma) = O(k n^2) \) and \( T(\gamma) = O(n \log n / \log k) \). It also has \( M(\gamma) = O(k n^2) \) and \( T(\gamma) = O(n \log n / \log k) \).

ADDITIONAL TOPICS

So far in this chapter we have aimed at characterizing what we judge to be the core aspects of the design and analysis of distributed algorithms. But progress has also been made along several other important avenues that are relevant from both a theoretical and an applications-related perspective. We devote this section to highlighting some of them.

Theoretical Aspects

Lower Bounds and Impossibility Results

An important issue that has persisted through the entire development of the field of distributed algorithms has been that of identifying what can and what cannot be computed distributedly, while asserting the minimum message and time complexities of the feasible computations whenever possible. Establishing such results depends on several factors, including the model of distributed computing that is assumed, the system’s topology, the existence of distinct identifications for all nodes, and the possibility of failures. We refer the reader to the survey article by Fich and Ruppert (2003) for a comprehensive treatment.

Computing on Anonymous Systems

Anonymity means that the assumption of distinct node identifications does not hold. Although it may be argued that studying distributed computations on anonymous systems has in the past had little practical impact, assuming anonymity is theoretically interesting because it highlights the influence of model- and topology-related characteristics on the possibilities of distributed computing. Electing a leader, for example, is naturally expected to be impossible under anonymity. What is curious, though, is that other problems that do not explicitly rely on node identifications also exist that cannot be solved distributedly if the system is anonymous. Several of the pertinent results refer to the consistent computation of functions from distributed inputs (Attiya, Snir, and Warmuth 1988; Attiya and Snir 1991; Yamashita and Kameda 1996), but there has also been recent interest in the impact of anonymity on other distributed tasks (e.g., routing in wireless sensor networks; cf. Dutra and Barbosa 2006).

Topological Awareness

Whenever a distributed algorithm is given for a graph \( G \) of unrestricted topology, the prevailing assumption in the field is that no node has built-in information on the structure of \( G \). Such information can come only at the expense of messages and time, and this has to be weighed against any complexity-related advantage one may expect from knowing it locally. One interesting line of research has addressed the question of how to give nodes built-in information that, while still essentially local, has the potential of providing the desired topological awareness. Flocchini, Mans, and Santoro (2003) survey the successful approach known as sense of direction, which is based on labeling edges (locally at each node) in a globally consistent manner.

Self-Stabilization

Throughout the chapter we have concentrated solely on systems that do not fail. The study of failures, however, is an important part of the field of distributed algorithms. One aspect of it is the design and analysis of asynchronous algorithms that, regardless of how local variables are initialized, necessarily converge to some global state satisfying a preestablished property. These are then infinite computations that require no initialization and are resilient to local-state corruption to a certain degree. The characteristic of necessary convergence to a desired global state is called “self-stabilization.” Its original introduction and also the first known examples are due to Dijkstra (1974); since then, the field has undergone considerable development (Dolev 2000).

Consensus Problems

These problems have a paradigmatic status in the study of failure-prone systems and ask that all nodes that do not fail reach some sort of agreement. In the asynchronous model, and considering node failures of the crash type only, the very influential study of Fischer, Lynch, and Paterson (1985) establishes the impossibility of agreement even if communication is fully reliable. In the synchronous model, by contrast, agreement can be reached even if we allow for the malicious (so-called Byzantine) failures in which nodes may fail by behaving unreliably instead of crashing (Pease, Shostak, and Lamport 1980; Lamport, Shostack, and Pease 1982), provided at least \( 2n/3 + 1 \) of the \( n \) nodes do not fail. The reason for such a wide gap between what happens in the two models is that failure detection is trivial in the synchronous model but unattainable in the asynchronous model. However, in many real-world systems that do not adhere to the synchronous model,
reaching agreement seems to be a definite possibility, as in fact demonstrated by the known solutions that adopt some sort of partially synchronous model or randomization (Gärtner 1999). The current framework within which these issues are studied was inaugurated by the work of Chandra and Toueg (1996) and its precursors by the same authors, and is centered on the notion of a failure detector: an imperfect distributed oracle that embodies the desired assumptions of partial synchronism. An introduction to the subject has been given by Raynal (2005).

Correctness Proofs
Similarly to what happens in all algorithm-oriented research areas, many researchers in the field of distributed algorithms have striven for the definition of frameworks that may ultimately lead to automatic correctness proofs. Input/output (I/O) automata and the various abstract structures through which they are combined are an example (Lynch et al. 1994). Another example is UNITY, originally introduced by Chandy and Misra (1988), comprising a programming notation and an associated logic. Ongoing progress on UNITY can be tracked through the PSP Group at UT Austin (2003), for example.

A Unifying Theory
As we generalize the synchronous and asynchronous models to encompass the possibility of failures and also communication by shared memory, the question of whether a unifying theory of computability and complexity for distributed computing can be constructed arises naturally. The surveys by Lamport and Lynch (1990) and Gafni (1999) offer two views of the issue separated by nearly a decade. A reasonable conclusion from the evolution of the field in the meantime between the two surveys seems to be that a theory has begun to emerge that focuses on the properties of complete problems—in a sense similar to that in which certain decision problems are NP-complete—using the tools of combinatorial topology (cf. Rajsbaum 2004 for the key references). Whether such a theory will be consolidated or remain a largely inchoate body of knowledge is yet to be seen.

Applications
Resource Sharing
Resource-sharing computations such as the one we used as an example above not only raise the safety-related issues (such as deadlock) of that example, but also are a source of important questions related to liveness issues and to the essential nature of synchronization procedures other than the synchronization for safety of the section on “Synchronizers” (Barbosa 2002). Answering these questions has led to the design of distributed schedulers targeted at deadlock- and lockout-free resource sharing. Such schedulers are rooted in the combinatorial properties that underlie all resource-sharing constraints, as can be seen, for example, in the seminal work of Chandy and Misra (1984) and in its further characterizations (Barbosa and Gafni 1989) and generalizations (Barbosa, Benevides, and França 2001).

Computing on Wireless Networks
The current ubiquity of wireless networks, including mobile ad hoc and sensor networks, has prompted researchers in the field to a whole new class of challenges. Depending on the nodes’ degree of mobility in the network, even the seemingly simple task of determining their locations has required careful attention and resulted in the development of a special theory (Aspnes et al. 2006) to support the creation of location-independent algorithms, for example like the one for routing given by Fonseca et al. (2005). In a similar vein, for many of the envisaged applications of sensor networks it is expected that nodes will have to synchronize their local views of real time, and again special distributed algorithms, for which the works of Su and Akyildiz (2005) and Fan and Lynch (2006) provide the bases, will be needed. Of course, what prevents the direct use of the algorithms we have seen throughout the chapter is the dynamic character of G’s topology. In such circumstances, even well-understood problems like determining a minimum-weight spanning tree distributively may be impossible to solve exactly, which has motivated the first steps toward a theory of distributed approximability (Elkin 2004). Further applications-related considerations, like noisy environments and severe memory and processing limitations, also seem to be bringing the issues of anonymity and self-stabilization back to the fore (Angluin et al. 2005, Stably computable properties of network graphs; Angluin et al. 2005, Self-stabilizing population protocols).

Computing on Complex Networks
Networks like the Internet (and also several others; cf. Bornholdt and Schuster 2003 and Newman, Barabási, and Watts 2006) are examples of what we call a “complex network,” that is, a large-scale, unstructured network of essentially unknown topology. Asynchronous algorithms designed to work on a generic graph G work also on such networks, but the networks’ massive scale may lead to unacceptable complexity figures in the case of some algorithms. However, since the discovery that the Internet may be an instance of a random graph with known degree distribution (Faloutsos, Faloutsos, and Faloutsos 1999), the possibility has been opened up of designing asynchronous algorithms that take such a distribution into account and produce less costly approximations of the algorithms for generic G. Recent examples include probabilistic approximations of dissemination by flooding (Stauffer and Barbosa 2007), with applications (Stauffer and Barbosa 2006a), and also the probabilistic algorithm of Stauffer and Barbosa (2006b) to approximate the construction of a spanner (Kortsarz and Peleg 1998).

CONCLUSION
This chapter has concentrated on discussing the field of distributed algorithms from two main perspectives. The first perspective has been that of how to handle the notion of time under full asynchronism, and consequently how to approach the treatment of global properties. The second perspective has been that of designing asynchronous algorithms up from its constituent building blocks. Our treatment has included a discussion of some fundamental building blocks and also examples of their use.
It is important for the reader to bear in mind that the field is much broader than our limited selection of topics may have implied. The main role of the section on “Additional Topics” has been to single out and highlight some of the topics we have not treated in depth but are equally interesting and important.

ACKNOWLEDGMENT
The author acknowledges partial support from CNPq, CAPES, and a FAPERJ BBP grant.

GLOSSARY
Asynchronous Algorithm: A distributed algorithm for the asynchronous model.
Asynchronous Model: The model of distributed computing in which nodes are driven by independent, local clocks and messages suffer unpredictable, though finite, delays.
Atomic Action: The uninterrupted first action of an initiator, or the uninterrupted response to the arrival of a message, during a run of an asynchronous algorithm.
Event: A sextuple characterizing each atomic action of a run of an asynchronous algorithm.
Execution: A set of events characterizing a run of an asynchronous algorithm.
Global State: A two-set partition of an execution that is consistent with the happened-before relation.
Global-State Recording Algorithm: An asynchronous algorithm to record a global state.
Happened-Before Relation: A binary relation that gives the causal precedence among the events of an execution.
Initiator: A node that initiates a run of an asynchronous algorithm.
Leader-Election Algorithm: A distributed algorithm that identifies a unique node on whose identification all nodes agree.
Message Complexity: The worst-case number of messages required during a run of a distributed algorithm.
Synchronizer: An asynchronous-algorithm fragment for transforming a synchronous algorithm into an asynchronous one.
Synchronous Algorithm: A distributed algorithm for the synchronous model.
Synchronous Model: The model of distributed computing in which all nodes operate in lockstep with a global clock and messages suffer delays that are bounded by the duration between successive clock pulses.
Time Complexity: The worst-case time required for completion of a run of a distributed algorithm.
Tree-Constructing Algorithm: A distributed algorithm that constructs a tree rooted on its initiator.

REFERENCES
Awerbuch, B. 1985. Complexity of network synchroniza-

CROSS REFERENCES