Chapter 14

Synchronizers

Don't walk behind me; I may not lead. Don't walk in front of me; I may not follow. Just walk beside me and be my friend. — Albert Camus

14.1 Introduction

The design of distributed algorithms is easier if we assume that the underlying network is synchronous rather than asynchronous. A prime example is that of computing a breadth-first search (BFS) tree in a network. Section 14.6 gives a simple algorithm to determine the breadth-first search tree in a graph, assuming a synchronous network. The corresponding problem on an asynchronous graph is more difficult. This motivates methods by which a synchronous network can be simulated by an asynchronous network. We show that, in the absence of failures, this is indeed possible using a mechanism called a synchronizer.

A synchronous network can be abstracted with the notion of a *pulse*. A pulse is a counter at each process with the property that any message sent in pulse i is received at pulse i. A synchronizer is simply a mechanism that indicates to a process when it can generate a pulse. In this chapter we will study synchronizers and their complexity.

To define properties of synchronizers formally, we associate a pulse number with each state s on the process. It is initialized to 0 for all processes. A process can go from pulse i to $i+1$ only if it knows that it has received all the messages sent during pulse i.

Given the notion of a pulse, the execution of a synchronous algorithm can be modeled as a sequence of pulses. In each pulse, a process first performs internal computation based on the messages received in the previous round, if any. After the computation, it sends messages to its neighbors as required by the application. Finally, it receives messages from neighbors that were sent in this round. It can execute the next pulse only when indicated by the synchronizer.

There are two aspects of the complexity of a synchronizer—the message complexity and the time complexity. The message complexity indicates the additional number of messages required by the synchronizer to simulate a synchronous algorithm on top of an asynchronous network. The time complexity is the number of time units required to simulate one pulse where a time unit is defined as the time required for an asynchronous message.

Some synchronizers have a nontrivial initialization cost. Let M_{init} be the number of messages and T_{init} be the time required for initialization of the synchronizer. Let M_{pulse} and T_{pulse} be the number of messages and time required to simulate one pulse of a synchronous algorithm. If a synchronous algorithm requires T_{synch} rounds and M_{synch} messages, then the complexity of the asynchronous protocol based on the synchronizer is given by:

$$M_{asynch} = M_{init} + M_{synch} + M_{pulse} * T_{synch}$$

$$T_{asynch} = T_{init} + T_{pulse} * T_{synch}$$

We model the topology of the underlying network as an undirected, connected graph. We assume that processes never fail. It is not possible to simulate a synchronous algorithm on an asynchronous network when processes can fail. In Chapter 15 we show that consensus is impossible in asynchronous systems when even a single process may fail. We also assume that all channels are reliable.

The notation used in this chapter is summarized in Figure 14.1.

N	The number of nodes in the network
E	The number of edges in the network
D	Diameter of the network

Figure 14.1: Notation

This chapter is organized as follows. Section 14.2 describes a simple synchronizer that requires exactly one message along each link in each direction per pulse. Section 14.3 describes a synchronizer called α that is optimal with respect to the time required for simulating a pulse but inefficient with respect to the message complexity. Section 14.4 describes a synchronizer called β that is efficient with respect to the message complexity but inefficient with respect to the time required for simulating a pulse. Section 14.5 presents a parameterized synchronizer that allows one to explicitly trade off the time complexity for the message complexity. Section 14.6 presents an application of synchronizers for designing a distributed algorithm for building the breadth-first search tree in an asynchronous network.

14.2 A Simple Synchronizer

A simple synchronizer can be built using the following rule: Every process sends exactly one message to all neighbors in each pulse. With this rule, a process can simply wait for exactly one message from each of its neighbors. To implement this rule, even if the synchronous algorithm did not require P_i to send any message to its neighbor P_j in a particular round, it must still send a "null" message to P_j. Furthermore, if the synchronous algorithm required P_i to send multiple messages, then these messages must be packed as a single message and sent to P_j.

The simple synchronizer generates the next pulse for process p at pulse i when it has received exactly one message sent during pulse i from each of its neighbors. The algorithm is shown in Figure 14.2.

The algorithm in 14.2 ensures that a process in pulse i receives only the messages sent in pulse i. If a process is in round i and it gets a message with pulse number $i+1$, then it buffers that message and waits for the message with pulse number i. Note that in an asynchronous network with the simple synchronizer, a process at pulse i can only receive messages sent during pulses i or pulse $i+1$. Thus instead of including the pulse number i with each message, it is sufficient to include a bit, i mod 2.

There is no special requirement for initialization of this synchronizer. When any process starts pulse 1, within D time units all other processes will also start pulse 1. Therefore, the complexity of initializing the simple synchronizer is

$$M_{init} = 0; \quad T_{init} = D.$$

Because each pulse requires a message along every link in both directions, we get the complexity of simulating a pulse as

$$M_{pulse} = 2E; \quad T_{pulse} = 1.$$

```
P_j::
 var
      pulse: integer initially 0;

 round i :
      pulse := pulse + 1;
      simulate the round i of the synchronous algorithm;
      send messages to all neighbors with pulse;
      wait for exactly one message from each neighbors with (pulse = i);
```

Figure 14.2: The implementation of a simple synchronizer at P_j

14.3 Synchronizer α

The synchronizer α is very similar to the simple synchronizer. We cover this synchronizer because it is a special case of a more general synchronizer γ that will be covered later. All the synchronizers discussed from now on are based around the concept of *safety* of a process. Process P is safe for pulse i if it knows that all messages sent from P in pulse i have been received.

The α synchronizer generates the next pulse at process P if all its neighbors are safe. This is because if all neighbors of P are safe then all messages sent to process P have been received.

To implement the α synchronizer, it is sufficient for every process to inform all its neighbors whenever it is safe for a pulse. How can a process determine whether it is safe? This is a simple matter if all messages are required to be acknowledged.

The complexity of synchronizer α is given below:

$$T_{init} = D; \qquad M_{init} = O(E)$$

$$T_{pulse} = O(1); \qquad M_{pulse} = O(E)$$

14.4 Synchronizer β

Although the synchronizers discussed so far appear to be efficient, they have high message complexity when the topology of the underlying network is dense. For large networks, where every node may be connected to a large number of nodes, it may be impractical to send a message to all neighbors in every pulse. The message complexity can be reduced at the expense of time complexity as illustrated by the β synchronizer.

The β synchronizer assumes the existence of a rooted, spanning tree in the network. A node in the tree sends a message *subtree-safe* when all nodes in its subtree are safe. When the root of the tree is safe and all its children are safe, then we can conclude that all nodes in the tree are safe. Now a simple broadcast of this fact via a *pulse* message can start the next pulse at all nodes. The broadcast can be done using the rooted spanning tree. Thus this algorithm simply uses the *convergecast* and *broadcast* algorithms discussed in Chapter **??**.

The initialization phase of this synchronizer requires a spanning tree to be built. This can be done using $O(N \log N + E)$ messages and $O(N)$ time. For each pulse, we require messages only along the spanning tree. Thus the message complexity for each pulse is $O(N)$. Each pulse also takes time proportional to the height of the spanning tree, which in the worst case is $O(N)$. In summary, the complexity of the β

synchronizer is

$$T_{init} = O(N); \qquad M_{init} = O(N \log N + E)$$
$$T_{pulse} = O(N); \qquad M_{pulse} = O(N).$$

14.5 Synchronizer γ

We have seen that the α synchronizer takes $O(1)$ time unit but has high message complexity $O(E)$ and the β synchronizer has low message complexity $O(N)$ but requires $O(N)$ time per pulse. We now describe the γ synchronizer that is a generalization of both α and β synchronizers. It takes a parameter k such that when k is $N-1$, it reduces to the α synchronizer and when k is 2 it reduces to the β synchronizer.

The γ synchronizer is based on *clustering*. In the initialization phase, the network is divided into clusters. Within each cluster the algorithm is similar to the β synchronizer and between clusters it is similar to the α synchronizer. Thus each cluster has a cluster spanning tree. The root of the cluster spanning tree is called the cluster leader. We say that two clusters are neighboring if there is an edge connecting them. For any two neighboring clusters, we designate one of the edges as the *preferred* edge.

The algorithm works as follows. There are two phases in each pulse. In both the phases, the messages first travel upward in the cluster tree and then travel downward. The goal of the first phase is to determine when the cluster is safe and inform all cluster nodes when it is so. In this phase, *subtree safe* messages first propagate up the cluster tree. When the root of the cluster gets messages from all its children and it is safe itself, it propagates the *cluster safe* message down the cluster tree. This phase corresponds to the β synchronizer running on the cluster. We also require that the nodes that are incident on preferred edges also send out *our cluster safe (ocs)* over preferred edges.

The goal of the second phase is to determine whether all neighboring clusters are safe. In this sense, it is like an α synchronizer. It uses additional two message types: *neighboring cluster safe (ncs)* and *pulse*. When a leaf in the cluster tree receives the *our cluster safe* message from all preferred edges incident to it, it sends *ncs* to its parent. Now consider an internal node in the cluster tree that has received *ncs* messages from all its children and has received *ocs* on all preferred edges incident to it. If it is not the cluster leader, then it propagates the *ncs* message upward; otherwise, it broadcasts the *pulse* message in its group.

For any clustering scheme c, let E_c denote the number of tree edges and preferred edges and H_c denote the maximum height of a tree in c. The complexity of the γ synchronizer is given by

$$M_{pulse} = O(E_c)$$

$$T_{pulse} = O(H_c)$$

The following theorem shows that any graph can be decomposed into clusters so that there is an appropriate trade-off between the cluster height and the number of tree and preferred edges.

Theorem 14.1 *For each k in the range $2 \leq k < N$, there exists a clustering c such that $E_c \leq kN$ and $H_c \leq \log N / \log k$.*

Proof: We give an explicit construction of the clustering. In this scheme, we add clusters one at a time. Assume that we have already constructed r clusters and there are still some nodes left that are not part of any cluster. We add the next cluster as follows.

Each cluster C consists of multiple layers. For the first layer, any node that is not part of the clusters so far is chosen. Assume that i layers ($i \geq 1$) of the cluster C have already been formed. Let S be the set of neighbors of the node in layer i that are not part of any cluster yet. If the size of S is at least $(k-1)$ times the size of C, then S is added as the next layer of the cluster C; otherwise, C's construction is finished.

Let us compute H_c and E_c for this clustering scheme. Since each cluster with level i has at least k^{i-1} nodes, it follows that H_c is at most $\log N / \log k$. E_c has two components—tree edges and preferred edges. The tree edges are clearly at most N. To count the preferred edges, we charge a preferred edge between two clusters to the first cluster that is created in our construction process. Note that for a cluster C, its construction is finished only when there are at most $(k-1)|C|$ neighboring nodes. Thus, for the cluster C, there can be at most $(k-1)|C|$ preferred edges charged to it. Adding up the contribution from all clusters, we get that the total number of preferred edges is at most $(k-1)N$.

■

14.6 Application: BFS Tree Algorithm

Assume that we are given a distinguished node v and our job is to build a breadth-first search tree rooted at v. A synchronous algorithm for this task is quite simple. We build the tree level by level. The node v is initially at level 0. A node at level i is required to send messages to its neighbor at pulse i. A process that receives one or more of these messages, and does not have a level number assigned yet, chooses the source of one of these messages as its parent and assigns itself level number $i+1$. It is clear that if the graph is connected, then every node will have its level number assigned in at most D pulses assuming that any message sent at pulse i is received at pulse $i+1$.

To simulate the synchronous algorithm on an asynchronous network, all we need is to use one of the synchronizers discussed in this chapter.

14.7 Problems

14.1. Give the pseudo-code for α, β, and γ synchronizers.

14.2. What is the message complexity of the asynchronous algorithm to construct a breadth-first search tree when it is obtained by combining the synchronous algorithm with (1) the α synchronizer, (2) the β synchronizer, and (3) the $\gamma(k)$ synchronizer?

14.3. Show how synchronizers can be used in a distributed algorithm to solve a set of simultaneous equations by an iterative method.

*14.4. **(due to [Awe85])** Give a distributed algorithm to carry out the clustering used by the γ synchronizer.

*14.5. **(due to [Lub85])** Let $G = (V, E)$ be an undirected graph corresponding to the topology of a network. A set $V' \subseteq V$ is said to be *independent* if there is no edge between any two vertices in V'. An independent set is *maximal* if there is no independent set that strictly contains V'. Give a distributed synchronous randomized algorithm that terminates in $O(\log |V|)$ rounds. Also, give an algorithm that works on asynchronous networks.

14.8 Bibliographic Remarks

The concept of synchronizers, and the synchronizers α, β, and γ were introduced by Awerbuch [Awe85]. The k-session problem and the lower bound on the time complexity of any asynchronous algorithm that solves the k-session problem is due to Arjomandi, Fischer, and Lynch [AFL83]. The reader is referred to the book by Raynal and Helary [RH90] for more details on synchronizers.