Chapter 7

Synchronizers

Synchronizers simulate an execution of a failure-free synchronous system in a failure-free asynchronous system. See [AW04, Chapter 11] or [Lyn96, Chapter 16] for a detailed (and rigorous) presentation.

7.1 Definitions

Formally, a synchronizer sits between the underlying network and the processes and does one of two things:

- A **global synchronizer** guarantees that no process receives a message from round r until *all processes* have sent their messages for round r.

- A **local synchronizer** guarantees that no process receives a message from round r until *all of that process's neighbors* have sent their messages for round r.

In both cases, the synchronizer packages all the incoming round r messages m for a single process together and delivers them as a single action `recv`(p, m, r). Similarly, a process is required to hand over all of its outgoing round-r messages to the synchronizer as a single action `send`(p, m, r)—this prevents a process from changing its mind and sending an extra round-r message or two. It is easy to see that the global synchronizer produces executions that are effectively indistinguishable from synchronous executions, assuming that a synchronous execution is allowed to have some variability in exactly when within a given round each process does its thing. The local synchronizer only guarantees an execution that is locally indistinguishable from an execution of the global synchronizer: an individual process can't tell the difference, but comparing actions at different (especially widely separated) processes may reveal some process finishing round $r + 1$ while others are still stuck in round r or earlier. Whether this is good enough depends on what you want: it's bad for coordinating simultaneous missile launches, but may be just fine for adapting a synchronous message-passing algorithm (as with distributed breadth-first search as described in §4.3) to an asynchronous system, if we only care about the final states of the processes and not when precisely those states are reached.

Formally, the relation between global and local synchronization is described by the following lemma:

Lemma 7.1.1. *For any schedule S of a locally synchronous execution, there is a schedule S' of a globally synchronous execution such that $S|p = S'|p$ for all processes p.*

Proof. Essentially, we use the same **happens-before** relation as in Chapter 6, and the fact that if a schedule S' is a causal shuffle of another schedule S (i.e., a permutation of T that preserves causality), then $S'|p = S|p$ for all p (Lemma 6.1.1).

Given a schedule S, consider a schedule S' in which the events are ordered first by increasing round and then by putting all sends before receives. This ordering is consistent with $\underset{S}{\Rightarrow}$, so it's a causal shuffle of S and $S'|p = S|p$. But it is globally synchronized, because no round r operation ever happens before a round $(r - 1)$ operation. □

7.2 Implementations

Here we describe several implementations of synchronizers. All of them give at least local synchrony. One of them, the beta synchronizer (§7.2.2), also gives global synchrony.

The names were chosen by their inventor, Baruch Awerbuch [Awe85]. The main difference between them is the mechanism used to determine when round-r messages have been delivered.

In the **alpha synchronizer**, every node sends a message to every neighbor in every round (possibly a dummy message if the underlying protocol doesn't send a message); this allows the receiver to detect when it's gotten all its round-r messages (because it expects to get a message from every neighbor) but may produce huge blow-ups in message complexity in a dense graph.

In the **beta synchronizer**, messages are acknowledged by their receivers (doubling the message complexity), so the senders can detect when all of their messages are delivered. But now we need a centralized mechanism to collect this information from the senders and distribute it to the receivers, since any particular receiver doesn't know which potential senders to wait for. This blows up time complexity, as we essentially end up building a global synchronizer with a central leader.

The **gamma synchronizer** combines the two approaches at different levels to obtain a trade-off between messages and time that depends on the structure of the graph and how the protocol is organized.

Details of each synchronizer are given below.

7.2.1 The alpha synchronizer

The alpha synchronizer uses local information to construct a local synchronizer. In round r, the synchronizer at p sends p's message (tagged with the round number) to each neighbor p' or $\mathsf{noMsg}(r)$ if it has no messages. When it collects a message or noMsg from each neighbor for round r, it delivers all the messages. It's easy to see that this satisfies the local synchronization specification.

This produces no change in time but may drastically increase message complexity because of all the extra noMsg messages flying around. For a synchronous protocol that runs in T rounds with M messages, the same protocol running with the alpha synchronizer will still run in T time units, but the message complexity will go up to $T \cdot |E|$ messages, or worse if the original algorithm doesn't detect termination.

7.2.2 The beta synchronizer

The beta synchronizer centralizes detection of message delivery using a rooted directed spanning tree (previously constructed). When p' receives a round-r message from p, it responds with $\mathsf{ack}(r)$. When p collects an ack for all the messages it sent plus an OK from all of its children, it sends OK to its parent. When the root has all the ack and OK messages it is expecting, it broadcasts go. Receiving go makes p deliver the queued round-r messages.

This works because in order for the root to issue go, every round-r message has to have gotten an acknowledgment, which means that all round-r messages are waiting in the receivers' buffers to be delivered. For the beta synchronizer, message complexity for one round increases slightly from M to $2M + 2(n - 1)$, but time complexity goes up by a factor proportional to the depth of the tree.

7.2.3 The gamma synchronizer

The gamma synchronizer combines the alpha and beta synchronizers to try to get low blowups on both time complexity and message complexity. The essential idea is to cover the graph with a spanning forest and run beta within each tree and alpha between trees. Specifically:

- Every message in the underlying protocol gets acked (including messages that pass between trees).
- When a process has collected all of its outstanding round-r acks, it sends OK up its tree.
- When the root of a tree gets all acks and OK, it sends ready to the roots of all adjacent trees (and itself). Two trees are adjacent if any of their members are adjacent.
- When the root collects ready from itself and all adjacent roots, it broadcasts go through its own tree.

As in the alpha synchronizer, we can show that no root issues go unless it and all its neighbors issue ready, which happens only after both all nodes in the root's tree and all their neighbors (some of whom might be in adjacent trees) have received acks for all messages. This means that when a node receives go it can safely deliver its bucket of messages.

Message complexity is comparable to the beta synchronizer assuming there aren't too many adjacent trees: $2M$ messages for sends and acks, plus $O(n)$ messages for in-tree communication, plus $O(E_{\text{roots}})$ messages for root-to-root communication. Time complexity per synchronous round is proportional to the depth of the trees: this includes both the time for in-tree communication, and the time for root-to-root communication, which might need to be routed through leaves.

In a particularly nice graph, the gamma synchronizer can give costs comparable to the costs of the original synchronous algorithm. An example in [Lyn96] is a ring of k-cliques, where we build a tree in each clique and get $O(1)$ time blowup and $O(n)$ added messages. This is compared to $O(n/k)$ time blowup for the beta synchronizer and $O(k)$ message blowup (or worse) for the alpha synchronizer. Other graphs may favor tuning the size of the trees in the forest toward the alpha or beta ends of the spectrum, e.g., if the whole graph is a clique (and we didn't worry about contention issues), we might as well just use beta and get $O(1)$ time blowup and $O(n)$ added messages.

7.3 Applications

See [AW04, §11.3.2] or [Lyn96, §16.5]. The one we have seen is distributed breadth-first search, where the two asynchronous algorithms we described in Chapter 4 were essentially the synchronous algorithms with the beta and alpha synchronizers embedded in them. But what synchronizers give us in general is the ability to forget about problems resulting from asynchrony provided we can assume no failures (which may be a very strong assumption) and are willing to accept a bit of overhead.

7.4 Limitations of synchronizers

Here we show some lower bounds on synchronizers, justifying our previous claim that failures are trouble and showing that global synchronizers are necessarily slow in a high-diameter network.

7.4.1 Impossibility with crash failures

These synchronizers all fail badly if some process crashes. In the α synchronizer, the system slowly shuts down as a wave of waiting propagates out from the dead process. In the β synchronizer, the root never gives the green light for the next round. The γ synchronizer, true to its hybrid nature, fails in a way that is a hybrid of these two disasters.

This is unavoidable in the basic asynchronous model, although we don't have all the results we need to prove this yet. The idea is that if we are in a synchronous system with crash failures, it's possible to solve **agreement**, the problem of getting all the processes to agree on a bit (see Chapter 9). But it's not possible to solve this problem in an asynchronous system with even one crash failure (see Chapter 11). Since a synchronous-with-crash-failure agreement protocol on top of a fault-tolerant synchronizer would give a solution to an unsolvable problem, the element of this stack that we don't know an algorithm for must be the one we can't do. Hence there are no fault-tolerant synchronizers.

We'll see more examples of this trick of showing that a particular simulation is impossible because it would allow us to violate impossibility results later, especially when we start looking at the strength of shared-memory objects in Chapter 19.

7.4.2 Unavoidable slowdown with global synchronization

The **session problem** [AFL83] gives a lower bound on the speed of a global synchronizer, or more generally on any protocol that tries to approximate synchrony in a certain sense. Recall that in a global synchronizer, our goal is to produce a simulation that looks synchronous *from the outside*; that is, that looks synchronous to an observer that can see the entire schedule. In contrast, a local synchronizer produces a simulation that looks synchronous *from the inside*—the resulting execution is indistinguishable from a synchronous execution to any of the processes, but an outside observer can see that different processes execute different rounds at different times. The global synchronizer we've seen takes more time than a local synchronizer; the session problem shows that this is necessary.

In our description, we will mostly follow [AW04, §6.2.2].

A solution to the session problem is an asynchronous protocol in which each process repeatedly executes some **special action**. Our goal is to guarantee that these special actions group into s **sessions**, where a session is an interval of time in which every process executes at least one special action. We also want the protocol to terminate: this means that in every execution, every process executes a finite number of special actions.

A synchronous system can solve this problem trivially in s rounds: each process executes one special action per round. For an asynchronous system, a lower bound of Attiya and Mavronicolas [AM94] (based on an earlier bound of Arjomandi, Fischer, and Lynch [AFL83], who defined the problem in a slightly different communication model), shows that if the diameter of the network is D, any solution to the s-session problem takes $(s-1)D$ time or more in the worst case. The argument is based on reordering events in any synchronous execution that takes less time to produce fewer than s sessions, using the happens-before relation described in Chapter 6.

We now give an outline of the proof that this is expensive. (See [AW04, §6.2.2] for the real proof.)

Fix some algorithm A for solving the s-session problem, and suppose that its worst-case time complexity is $(s-1)D$ or less. Consider some synchronous execution of A (that is, one where the adversary scheduler happens to arrange the schedule to be synchronous) that takes $(s-1)D$ rounds or less. Divide this execution into two segments: an initial segment γ that includes all rounds with special actions, and a suffix δ that includes any extra rounds where the algorithm is still floundering around. We will mostly ignore δ, but we have to leave it in to allow for the possibility that whatever is happening there is important for the algorithm to work (say, to detect termination).

We now want to perform a causal shuffle on γ that leaves it with only $s-1$ sessions. Because causal shuffles don't affect time complexity, this will give us a new bad execution $\gamma'\delta$ that has only $s-1$ sessions despite taking $(s-1)D$ time.

The first step is to chop γ into $s-1$ segments $\gamma_1, \gamma_2, \ldots \gamma_{s-1}$ of at most D rounds each. Because a message sent in round i is not delivered until round $i+1$, if we have a chain of k messages, each of which triggers the next, then if the first message is sent in round i, the last message is not delivered until round $i+k$. If the chain has length D, its events (including the initial send and the final delivery) span $D+1$ rounds $i, i+1, \ldots, i+D$. In this case the initial send and final delivery are necessarily in different segments γ_i and γ_{i+1}.

Now pick processes p and q at distance D from each other. Then any chain of messages starting at p within some segment reaches q after the end of the segment. It follows that for any events e_p of p and e_q of q in the *same* segment γ_i, $e_p \underset{\gamma\delta}{\not\Rightarrow} e_q$. So there exists a causal shuffle of γ_i that puts all events of p after all events of q.[1] By a symmetrical argument, we can similarly put all events of q in a segment after all events of p in the same segment. In both cases the resulting schedule is indistinguishable by all processes from the original.

So now we apply these shuffles to each of the segments γ_i in alternating order: p goes first in the odd-numbered segments and q goes first in the odd-numbered segments. Let's write the shuffled version of γ_i as $\alpha_i\beta_i$ for odd i and $\beta_i\alpha_i$ for even i; in each case, α_i contains only events of p and other processes that aren't q and β_i contains only events of q and other processes that aren't p.

When we put these alternating shuffles together, we get an execution that looks like this example with $s-1=4$:

$$\alpha_1\beta_1\beta_2\alpha_2\alpha_3\beta_3\beta_4\alpha_4\delta$$

Now let's count sessions. Since a session includes special actions by both p and q, it can't lie entirely within α intervals or β intervals. contains only steps of p and other processes that aren't q or an interval that contains only steps of q and other processes that aren't p. So any session has to span one of the points in the schedule marked by slashes below:

$$\alpha_1/\beta_1\beta_2/\alpha_2\alpha_3/\beta_3\beta_4/\alpha_4\delta$$

[1] Proof: Because $e_p \underset{\gamma\delta}{\not\Rightarrow} e_q$, we can add $e_q < e_p$ for all events e_q and e_p in γ_i and still have a partial order consistent with $\underset{\gamma\delta}{\Rightarrow}$. Now apply topological sort to get the shuffle.

There is one such point for each of our original $s-1$ intervals, so we get at most $s-1$ sessions.

This means that any algorithm that runs in time $(s-1)D$ in the worst case (here, the original synchronous execution) can't guarantee to give s sessions in all cases (it fails in the shuffled asynchronous execution). Note that this is not quite the same as saying that any execution with at least s sessions must take $(s-1)D$ time. Instead, we've shown that algorithm that guarantees we get at least s sessions sometimes takes more than $(s-1)D$ time, even though it might sometimes use less time if it gets lucky.