Contents

6 Simulating Synchronous Systems

Chapter Contents

Learning Goals

The main goal of this chapter is to introduce a simple yet highly idealized model of distributed computing: the synchronous model. We design and analyze algorithms for fundamental problems in the synchronous model, and discuss the model's power and limitations. We then show how one can simulate the synchronous model in an asynchronous system.

6.1 Overview

Perhaps the simplest theoretical distributed models are synchronous models. Here, we describe a *synchronous message passing* model, SMP, that abstracts away many computational details that must be considered in practice: local computational costs, bandwidth restrictions, asynchrony, and faults. Nonetheless, SMP is instructive for designing and reasoning about distributed protocols. In SMP, all nodes progress in lock step, and all messages sent in each round are received before the next round begins. Thus, all nodes progress in a coordinated way. The only computational restriction in SMP is *locality*: each node maintains a private state, and can only communicate with neighboring nodes to learn about their states.

Since the synchronous model assumes so few restrictions on its computational power, lower bounds for this model transfer *a fortiori* to weaker, more realistic computational models. Yet synchronous protocols can often be adapted to more realistic models as well—the challenge is to understand how.

We next introduce a *fully asychronous message passing* model, AMP, that makes no assumptions whatsoever about synchrony between nodes. In AMP, all computations occur in response to *events* without any synchrony between the timing of events; local computations and message delivery may take arbitrarily long to complete.

Despite the seemingly pessimistic assumption of total asynchrony in the AMP model, we present a general technique, called an α*-synchronizer*, that allows one to simulate any algorithm for the SMP model in the seemingly much weaker AMP model. Specifically, we will prove the following theorem.

Theorem 6.18. *Given any algorithm $\mathcal{B}$ in SMP, there is an algorithm $\mathcal{A}$ that simulates $\mathcal{B}$ in AMP. The asynchrononous time complexity of simulating the first T rounds of $\mathcal{B}$ is $t_0 + T$ in executions where all nodes $v \in V$ have woken up by time t_0 and for all $i \in \mathbb{N}_{>0}$ the event of v receiving $\gamma_{v,i}^{\mathcal{B}}$ happens by time $t_0 + i$. In graphs of diameter D, this implies an asynchronous time complexity of $D + T$ for simulating the first T rounds of $\mathcal{B}$ in such executions.*

In general, it is often fruitful to first understand how to *simulate* a more powerful model before attempting to solve a problem in the more restrictive model directly. The α-synchronizer technique we describe gives a general procedure for simulating executions of SMP in AMP.

6.2 Synchronous Message Passing

We describe a distributed system by a simple, connected graph $G = (V, E)$ (see Appendix **??**), where V is the set of $n := |V|$ nodes (our computational entities,

e.g., clock domains on a chip, processors in a multi-core machine, or computers in a network). Nodes v and w can directly communicate if and only if there is an edge $\{v, w\} \in E$. We model each node in the network as a finite state machine, where the state of a node includes the contents of messages received from other nodes .

Definition 6.1. *[The SMP model] The network is modeled as a simple graph $G = (V, E)$ of n nodes. Each node $v \in V$ has a unique* port number *assigned to each neighbor from the range $1, 2, \ldots, \deg(v)$ (where $\deg(v)$ denotes the* degree*—i.e., number of neighbors—of v). Nodes are modeled by finite state machines $T_v = (S, s_{v,0}, \Sigma, \Lambda, t, o)$, where the node-dependent initial state $s_{v,0}$ is referred to as the* input *of node v. The input and output alphabets are of the form $\Sigma = \Sigma_0^{\deg(v)} \times \Gamma$ and $\Lambda = \Sigma_0^{\deg(v)} \times \Omega$, where Σ_0 is the set of admissible messages, Γ is the set of possible local inputs, and Ω is the set of local outputs.* [1] *Σ_0 contains the special symbol $\perp$, which will be a shorthand for "no message," and Γ and Ω contain the symbol to allow for indicating no local input or output, respectively. Set $m_{v,p,0} := \perp$ and $\gamma_{v,0} := \perp$ for all $p \in \{1, 2, \ldots, \deg(v)\}$. An algorithm is executed in* synchronous rounds*, where in each round i, each node is given a local input $\gamma_{v,i} \in I$ and performs the following steps:*

1. *set $s_{v,i} := t(s_{v,i-1}, (m_{v,1,i-1}, m_{v,2,i-1}, \ldots, m_{v,\deg(v),i-1}, \gamma_{v,i}))$ to update its state;*
2. *for $p \in \{1, \ldots, \deg(v)\}$, send to the neighbor whose port number at v is p the message $o(s_{v,i-1}, (m_{v,1,i-1}, m_{v,2,i-1}, \ldots, m_{v,\deg(v),i-1}, \gamma_{v,i}))_p$;*
3. *receive messages, i.e., for $p \in \{1, \ldots, \deg(v)\}$ set $m_{v,i,p}$ to the message received from the neighbor whose port number at v is p; and*
4. *output $o(s_{v,i-1}, (m_{v,1,i-1}, m_{v,2,i-1}, \ldots, m_{v,\deg(v),i-1}, \gamma_{v,i}))_{\deg(v)+1}$.*

If in round i the FSM of v reaches a state s satisfying that

- *$t(s_{v,i}, \cdot) = s_{v,i}$ (no further state change possible),* [2]
- *$o(s_{v,i}, \cdot)_p = \perp$ for $p \in \{1, \ldots, \deg(v)\}$ (no further messages can be sent), and*
- *$o(s_{v,i}, \cdot)_{\deg(v)+1} = o_v$ for some $o_v \in \Omega$ (no further output change),*

[1] Note that this means that the state machine as a whole depends on $\deg(v)$. For simplicity, we omit this from the notation. To have each node run the exact same FSM, one could replace $\deg(v)$ by the maximum node degree, padding message tuples with $\perp$. If the algorithm utilizes knowledge of $\deg(v)$, it then needs to be given to v as part of $s_{v,0}$ or using Γ. The node-dependent initial states can also be used to model that different nodes might run different state machines, by encoding the "type" of the node in $s_{v,0}$.

[2] Here $\cdot$ stands for "arbitrary argument," i.e., $t(s_{v,i}, \cdot)$ is the function obtained from t by fixing its first argument to be $s_{v,i}$.

we say that v terminated in round i with output o_v. *and call* s *a* terminal state.

Remark 6.2. *In this book, we are mostly studying tasks that are repetitive or ongoing. Hence, most algorithms will never terminate. For "one-shot" tasks where termination occurs, there is the implicit assumption that the algorithm's state machine is part of a bigger state machine that manages the control flow beyond the specific task considered and provides the input to the node (if there is any). Single-use hardware that is studied from an algorithmic perspective is rare!*

Remark 6.3. *It is uncommon to specify algorithms in the SMP model as above for analysis purposes, as even the most conceptually simple algorithms become hard to read and understand this way. However, implementing the algorithm in hardware necessitates to translate the pseudocode or textual description into a state machine—or rather the corresponding circuit—as described above. If different nodes behave differently, to avoid blowing up the circuit size, one should use different state machines, rather than employing the hack of lumping everything together into a single state machine!*

In SMP, we think of the state of a node v at the beginning of round as consisting of both the state of v's "internal" finite state machine, as well as the state (i.e., value) of messages received in the previous round. Thus, for a node v with two neighbors, the state of v is a triple of values $(\text{STATE}_v, \text{REC}_1, \text{REC}_2)$. Here REC_1 and REC_2 are the messages recieved from v's neighbors with port numbers 1 and 2 (respectively) in the previous round. The transition function for v determines both how v updates STATE_v and the messages, SND_1 and SND_2, v sends to its neighbors in Step 2 of the current round.

Example 6.4 (Distributed Program Counter)**.** *In a* distributed program counter*, each node* v *maintains a variable* $\text{COUNT}_v \in \mathbb{N}$ *satisfying the following conditions:*

1. monotonicity*: for all times* t, t' *with* $t \leq t'$*, we have* $\text{COUNT}_v(t) \leq \text{COUNT}_v(t')$*;*
2. increment*: if* $\text{COUNT}_v(t) > 0$ *at time* t*, then there exists some time* $t' (< t)$ *for which* $\text{COUNT}_v(t') = \text{COUNT}_v(t) - 1$*;*
3. waiting for neighbors*: for all times* t *and all neighbors* w *of* v $\text{COUNT}_v(t) \leq \text{COUNT}_w(t) + 1$.

Conditions 1 and 2 simply require that COUNT_v *attains values* $0, 1, 2, \ldots$ *sequentially (though the count is allowed to stop at any time). Condition 3 is important for distributed coordination: it stipulates that* v *cannot increase* COUNT_v *until all of its neighbors' counters have also reached the value* COUNT_v.

Implementing a distributed program counting in SMP is extremely simple, as each node can set $\textsc{count}_v$ *to the current round:*

```
1: COUNT := 0;
2: for all rounds r do
3:     COUNT := COUNT + 1
4: end for
```

The pseudo-code above clearly satisfies the monotonicity and increment conditions. The wait-up condition is satisfied due to the specification of the SMP model: all nodes are assumed to transition from one round to the next at precisely the same time. Thus, in the SMP model, the pseudo-code above gives the stronger guarantee that for all times t *and all nodes* $v, w \in W$*, we have* $\textsc{count}_v(t) = \textsc{count}_w(t)$*. Moreover, this perfect synchrony is achieved without any communication! Achieving such perfect synchronization is, of course, impossible in practice. In the following section, we will show how to implement a distributed program counter in a fully asynchronous model. Understanding how to do so will give the key insight into simulating* any *synchronous algorithm in the asynchronous model.*

E6.1 Consider a relaxation of a distributed program counter where item 2 is replaced with:

2' *weak increment*: if $\textsc{count}_v(t) > \textsc{count}_v(0)$ at time t, then there exists some time $t'(< t)$ for which $\textsc{count}_v(t') = \textsc{count}_v(t) - 1$.

That is, $\textsc{count}_v$ may be initialized to an arbitrary count, but the other properties of a distributed program counter must be maintained. Write pseudo-code for an algorithm that solves this relaxed program counter problem, and prove its correctness (i.e., that it satisfies the three conditions of a distributed program counter with the modified condition 2).

In SMP, an important measure of the quality of an algorithm is the number of rounds the algorithm requires in order to perform some task. (Other important considerations might be the message size, number of states in the finite state machine, or complexity of local computations performed. For now, we focus on the number of rounds.) The notion of *time complexity* formalizes a measure of complexity for performing tasks in SMP.

Definition 6.5. *Let* $\mathcal{A}$ *be an algorithm in the SMP model, and let* $\mathcal{T}$ *denote a task performed by* $\mathcal{A}$*. The* time complexity *of* $\mathcal{A}$ *completing* $\mathcal{T}$ *is defined to be*

$$\sup\{r_1 - r_0 \mid \mathcal{T} \text{ initiated in round } r_0, \text{ completed in round } r_1\}, \tag{6.1}$$

where, for a fixed network G*, the supremum is taken over all possible states of the individual nodes in round* r_0*. That is, the time complexity of* $\mathcal{A}$ *completing*

$\mathcal{T}$ on G is the maximum number of rounds that elapse between when $\mathcal{T}$ is initiated until it is completed.

Remark 6.6. *We defined time complexity as the worst-case completion time for a fixed network G. However, it is often valuable to understand the performance of algorithms on families of graphs, and how the time complexity depends on different graph parameters. For example, one would often like to know how the performance of an algorithm scales with the size—i.e., number of nodes—of the network. One can then interpret the* sup *in (6.3) to be taken over all inputs and all networks with n nodes, so that the time complexity becomes a function n. Thus the statement "algorithm $\mathcal{A}$ has time complexity $O(n)$" can be interprated as saying that the worst case time complexity of $\mathcal{A}$ over all graphs with n nodes scales (at most) linearly in the size of the network. Similarly, one can analyze how the time complexity of $\mathcal{A}$ depends on, say, the network diameter, D, or maximum degree of a node, Δ (or even a combination of parameters) by interpretating the* sup *in (6.3) as taken over all networks with fixed values of these parameters, and treating the complexity as a function of the parameters of interest.*

Remark 6.7. *Definition 6.5 allows for the possibility that a task is completed in the same round that it is initiated. Thus, the time complexity for a task can be* 0*. In this case, the nodes immediately decide on their local outputs, without ever considering any input apart from their initial state!*

6.2.1 Example: Simultaneous Restart

Here, we introduce a fundamental problem in hardware design that we call *simultaneous restart*.[3] In the simultaneous restart problem, a node—which we call the *leader*—enters a state READY when it receives a local input indicating this, at the beginning of some arbitrary round r_0. The goal is for all nodes to simultaneously enter a state START at some later round $r_1 > r_0$.

For the present discussion, the network consists of a path of n nodes, which we will refer to as $v_1, v_2, \ldots, v_n$, ordered from left to right. For each v_i with $2 \leq i \leq n-1$, v_{i-1} is v_i's *left neighbor*, with associated port number 1, and v_{i+1} is v_i's *right neighbor* with port number 2. We assume that the left-most node v_1 is the leader.

[3] Historically, this problem was called the "firing squad" problem, but we opt for the less morbid and more descriptive "simultaneous restart." See the chapter notes for a discussion of the classical literature on the subject.

6.2.1.1 A Solution with Counters We first present a conceptually simple counter-based solution to the simultaneous restart problem, assuming that the network size, n, is known to the leader, and that messages may be as large as $\log n$ bits. While the solution we present is conceptually straightforward, the resources required to implement the counter-based solution in hardware may be unsuitably large. We present a solution where the state space and message sizes are constant (indpendent of n) in the sequel.

In our counter-based solution, the restart is initiated when the leader, v_1, receives a signal "RESTART." At that point, the leader starts a countdown of $n-1$ rounds until it enters the START state. It also decrements its counter to $n-2$, and sends this value to its right neighbor. Upon receiving a counter value, each v_i decrements the received value, sets its own counter to the decremented value, and sends this value to its right neighbor. Each round thereafter, v_i decrements its counter until it reaches 0. Once v_i's counter reaches 0, v_i transitions to START. We give pseudo-code in Algorithm 1.

Algorithm 1 RestartCounter.

```
1:  COUNT := ∞
2:  for all rounds r do
3:      if LEADER and receive RESTART then
4:          COUNT := n − 1                          ▷ set local counter
5:          SND_2 := n − 1                ▷ send counter value to right neighbor
6:      end if
7:      if REC_1 ≠ ∅ then                  ▷ receive a count from left neighbor
8:          COUNT := REC_1 − 1
9:          SND_2 := COUNT          ▷ send decremented count to right neighbor
10:     else if COUNT < ∞ then
11:         COUNT := COUNT − 1
12:     end if
13:     if COUNT = 0 then
14:         STATE := START
15:     end if
16: end for
```

Proposition 6.8. *The algorithm* RestartCounter *(Algorithm 1) solves the simultaneous restart problem in* $n-1$ *rounds.*

Proof. Let r_0 denote the round in which the leader receives the RESTART message. For $i = 0, 1, 2, \ldots, n-1$, we claim that the following properties hold at the end of round $r_0 + i$:

1. v_i sets COUNT_i to $n - i - 1$ and sends this value to v_{i+1},
2. all v_j with $j \leq i$ satisfy $\text{COUNT}_j = n - i - 1$.

We argue by induction on i. The base case $i = 0$ is immediate from Lines 4 and 5. For the inductive step, assume items 1 and 2 above hold for round $r_0 + i$ with $i > 0$. By the inductive hypothesis, node v_{i+1} receives the message "$n - i - 1$" from v_i in round $r_0 + i + 1$. In Lines 8 and 9, v_{i+1} sets COUNT to $n - i - 2 = n - (i + 1) - 1$ and sends this value to v_{i+2}. Thus v_i satisfies items 1 and 2 of the claim in round $r_0 + i + 1$. For each $j < i + 1$ the inductive hypothesis implies that $\text{COUNT}_j = n - i - 1$ at the beginning of round $i + 1$. This value is decremented in Line 11, so that $\text{COUNT}_j = n - (i + 1) - 1$ at the end of round $r_0 + i + 1$. Therefore v_j satisfies item 2 for all $j < i + 1$ at the end of round $i + 1$ as well. This concludes the inductive argument.

By item 2 above, all nodes transition to state START in round $r_1 = r_0 + i - 1$ (Line 14), and no node transitions to START in any round $r_0 + i$ with $0 \leq i < n - 1$. Therefore, RestartCounter solves the simultaneous restart problem, as desired. □

E6.2 Suppose that you can customize each state machine, i.e., the state machine of node v_i may depend on i. Modify Algorithm 1 to work with 1-bit messages. Prove that the modified algorithm is correct!

E6.3 Consider the case where the network $G = (V, E)$ is an arbitrary connected graph (i.e., not a path as described above). Suppose a fixed node v_1 is the leader, and that the radius R (i.e., the maximum distance from v_1 to any other node in the network) is known to v_1. Modify Algorithm 1 to solve the simultaneous restart problem on G, and prove the correctness of your algorithm. What is the time complexity of the modified algorithm?

6.2.1.2 A Solution with Constant-size FSMs Here we describe a solution to the simultaneous restart problem that requires no previous knowledge of the size of the network, and uses only a constant number of node states, independently of the network size. The basic idea of the solution is recursive: each "phase" divides the network in half, and then recursively solves the firing squad problem on the two (equal sized) halves of the network. More specifically, the first phase of the algorithm finds the midpoint(s) of the network, which serve as leader(s) for the second phase. In each phase, the size of each "active" component of the network gets cut in half, as does the amount of time to complete the phase. At the end of the last phase, all nodes are in the READY

state, at which point they simultaneously transition to START. Before the final phase, each node will have at least one neighbor who is not in the READY state, so no one transitions to START before the final phase.

Finding the center node of an "active" component—i.e., a set of nodes between consecutive READY nodes—is at the heart of the algorithm. The idea is simple: initially the leader sends two messages to its right neighbor: a "fast" message and a "slow" message. The fast message progresses one hop per round, until it reaches the right end of the network. At this point, the fast message bounces back from right to left, still moving at one hop per time step. Meanwhile, the slow message moves to the right at one hop every 3 rounds. Thus, after $\lfloor 3n/2 \rfloor$ rounds, the two messages meet at the middle node or adjacent middle nodes (depending on the parity of n). When the two messages meet, the node(s) where they meet enter(s) the state READY, and initiate(s) fast/slow messages both to the right and left. This concludes the first phase. The subsequent phases progress analogously with the newly READY nodes serving as leaders for the next phase.

In the k-th phase, the size of each active component is at most $n/2^k$, and all active components have the same size. In the final phase, all nodes are in the READY state, and they simultaneously transition to START in the following round. As for the run-time of the procedure, the duration of a phase is $3/2$ times the size of active components during that phase. Thus, the total run-time is

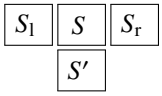
$$\frac{3}{2}n + \frac{3}{2} \cdot \frac{n}{2} + \frac{3}{2} \cdot \frac{n}{4} + \cdots < 3n \sum_{k=1}^{\infty} \frac{1}{2^k} = 3n.$$

In order to describe and analyze the algorithm sketched above formally, we describe explicit states and transition functions. The state space consists of 16 states, with the following symantics:

R the ready state (a.k.a. READY),
S the start state (a.k.a. START),
Q the quiescent state (i.e., the nominal state before the procedure begins),
Q' the quiescent state of the right-most node,
$\overrightarrow{F}_i, \overleftarrow{F}_i$ fast message states moving right and left, respectively, for $i = 1, 2$
$\overrightarrow{S}_i, \overleftarrow{S}_i$ slow message states moving right and left, respectively, for $i = 1, 2, 3$,
$\overrightarrow{W}, \overleftarrow{W}$ waiting states indicating the direction of the previous "message" state.

We assume that each node sends its internal state to its neighbors in each round. Thus, the transition function is specified by determining the next local state of each node from the local states of itself and its neighbors at the end of

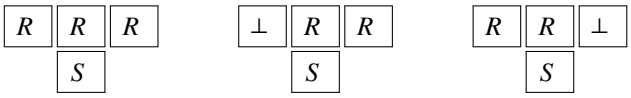
the previous round. We use the notation



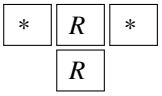
to indicate that a node in state S whose left and right neighbors are in states S_l and S_r, respectively, transititions to state S'.

We assume that for all rounds $r < r_0$, every node is in state Q except the right-most nodes, which is in state Q'. In round r_0, the leader v_0 transitions to the ready state R. We now describe the transition function, along with the semantics. We use the "wildcard" $*$ to indicate an arbitrary state. We use $\perp$ to denote that a neighbor is nonexistent (i.e., for the left neighbor of v_1 and the right neighbor of v_n. In order to simplify the presentation, the following rules are written in order of precedence. That is, if two rules apply (due to wildcards), then the first rule appearing on the list below is applied.

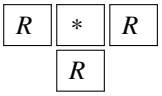
1. Start if everyone is ready



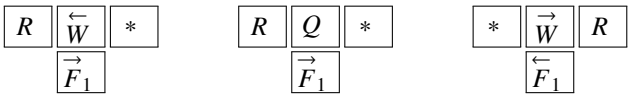
2. Ready stays ready



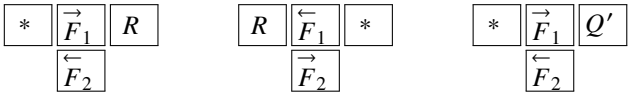
3. Ready if both neighbors are



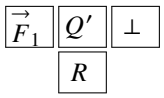
4. Ready states initiate a fast message in the correct direction



5. Fast messages bounce

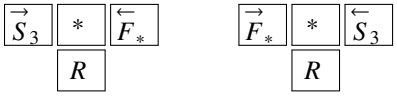


6. Rightmost node ready when first message arrives

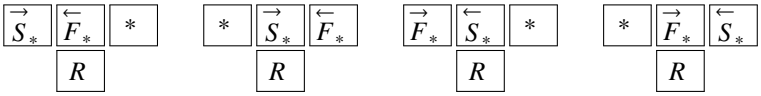


7. Start a slow message after fast message leaves

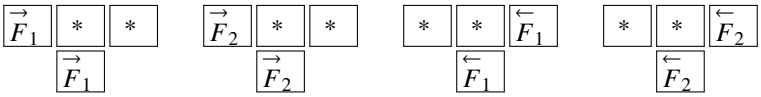
8. Pre-collision messages create ready nodes



9. Colliding messages create ready nodes

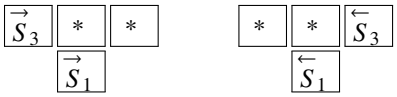


10. Fast messages move forward



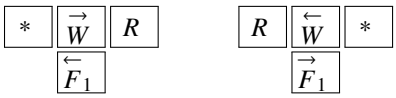
11. Slow messages wait

12. Slow messages move forward



13. Wait after sending a message

14. Newly ready nodes send fast messages



We give an illustration of an execution in Figure 6.1.

In order to prove the correctness of our construction, we formally introduce some terminology. We assume without loss of generality that $r_0 = 0$. We call an interval $I = [a, b]$ *active* in round r if v_{a-1} and v_{b+1} are state R or Q', but no v_i with $i \in I$ is in state R. We say that I *becomes active* in round r if I is an active interval in round r, but not in round $r - 1$.

Observation 6.9. *Suppose I becomes active in round $r > 0$. Then by Rule 2, in round $r - 1$ there was some active interval I' such that $I \subseteq I'$, and some node v_i with $i \in I'$ transitioned to state R in round r.*

We say that a *phase* begins in round r if some interval I becomes active in round r. By convention, the zeroth phase begins in round 0 when v_1 transitions to R. The *duration* of the phase is the number of rounds until another phase begins. In what follows, we will bound the duration of each phase and show that during each phase, all active intervals have the same size.

Lemma 6.10. *Suppose a phase begins in round $r > 0$, and the interval $I = [a, b]$ becomes active in round r. Let $c = \lfloor (a + b)/2 \rfloor$, $d = \lceil (a + b)/2 \rceil$, and $s = \lfloor 3\,|I|\,/2 \rfloor + 1$.*

Then the following hold.

1. *In round r, all nodes v_i with $i \in I$ are in the same state, either $\overleftarrow{W}$, $\overrightarrow{W}$, and nodes v_{a-1} and v_{b+1} are in state R.*
2. *I is active for all rounds $r, r + 1, \ldots, s - 1$.*
3. *If $|S| > 2$, then a new phase begins in round $r + s$ with intervals $I_\ell = [a, c - 1]$ and $I_r = [d + 1, b]$ becoming active.*
4. *If $|S| = 2$ then all nodes v_i with $i \in I$ transition to state R in round $r + s$.*

(Note: A proof sketch will be added later. You're not expected to verify this on your own.)

Proposition 6.11. *The 16 state machine described above solves the simultaneous restart problem on a network of size n in at most $3n$ rounds.*

Proof. For simplicity, we assume $n \geq 4$ and leave the cases $n = 2, 3$ as an exercise. We first prove that the state machine solves the simultaneous restart problem. To this end, observe that by Lemma 6.10, there is a sequence of

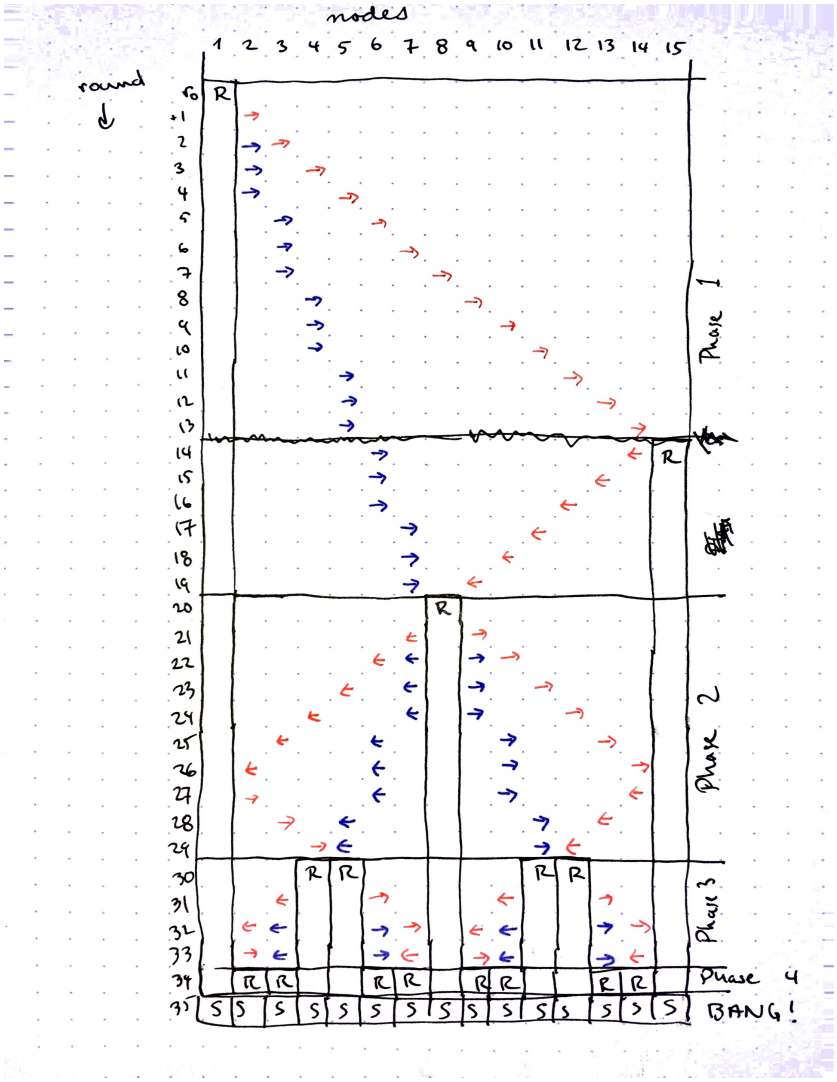


Figure 6.1
An illustration of the firing squad algorithm execution with $n = 15$ nodes.

rounds $r_0 < r_1 < \cdots$ such that a phase begins in each round r_i, and for each i, no phase begins in any round r satisfying $r_i < r < r_{i+1}$.

Claim. For every i and round $r \in [r_i, r_{i+1} - 1]$, all active intervals I have the same size, and all such intervals are deactivated in round r_{i+1}.

Proof of claim. We argue by induction on i. For the base case $i = 1$, there is a single active interval $I = [2, n-1]$ in round r_0. By Lemma 6.10, I is deactivated in round $r_1 = r_0 + s$.

For the inductive step, suppose the claim holds for the phase beginning in round r_{i-1}, and let $I_1, I_2, \ldots, I_m$ be the intervals active during this phase. Applying Lemma 6.10 and the inductive hypothesis, all I_j are the same size, and each I_j becomes deactivated in round $r_i = r_{i+1} + s$, where s is computed as in Lemma 6.10. Now fix some $I = I_j$, and let I_ℓ and I_r be the intervals computed in Lemma 6.10. Then by the same lemma $|I_\ell| = |I_r| = \lfloor (|I| - 1)/2 \rfloor$, and either both intervals are empty, or both become active in round r_i. Thus, the claim holds for r_i as well.

By the claim, all active intervals during the ith phase have the same length, which we denote ℓ_i. Moreover, by Lemma 6.10, we have

$$\ell_{i+1} = \lfloor (\ell_i - 1)/2 \rfloor. \tag{6.2}$$

Therefore, there exists some minimal k for which $\ell_k \leq 2$. At the end of the kth phase, all active nodes transition to state R by item 4 or Lemma 6.10. Finally, by Rule 1 of the transition function, all nodes simultaneous transition to S in the subsequent round. This concludes the proof of correctness.

To prove the run-time of the algorithm, by Lemma 6.10, duration of the ith phase is $\lfloor 3\ell_i/2 \rfloor \leq \frac{3}{2}\ell_i$, where $\ell_0 = n - 2 < n$. Applying Equation (6.2) inductively, we have $\ell_i < \ell_0/2^i$. Therefore, the total duration is at most

$$\begin{aligned} \sum_{i=0}^{k} \frac{3}{2}\ell_i &< \sum_{i=0}^{k} \frac{3}{2}\frac{\ell_0}{2^i} \\ &< \frac{3}{2} n \sum_{i=0}^{\infty} 2^{-i} \\ &= 3n. \end{aligned}$$

This gives the desired bound on the round complexity. □

E6.4 Prove Proposition 6.11 in the cases $n = 2, 3$.

E6.5 In the preceding discussion, we assumed that only the "leader" node v_1 could spontaneously initiate the execution of the algorithm. Modify the finite state

restart algorithm so that any single node can initiate a restart in a round. (Assume that only a single node initiates the restart during the execution.) What is the round complexity of the new algorithm?

6.3 Asynchronous Message Passing

As described above, the SMP model is unrealistic as a model for real computer systems, as it makes overly optimistic assumptions about the computational power of each node and synchrony between nodes in the network. Here, we present a model—the AMP model—that does not make any such assumptions. In the AMP model, nodes are not synchronized, and even simple local computations may take a long time to complete. The AMP model does not explicitly bound computational time, space, or communication, but these aspects can be reflected in the amount of time it takes for a processor to complete a single "step" of its computation.

Like the SMP model, nodes in the AMP model communicate by passing messages to neighboring nodes in the underlying network. Similar to the SMP model, the AMP model is fault-free: every local computation is faithfully (if slowly) performed, and all sent messages are eventually received by their intended recipient. Thus, the AMP model is meant to encapsulate all models that are based on fault-free message exchange.

Definition 6.12 (The AMP message passing model)**.** *The network is modeled as a simple graph $G = (V, E)$ of n nodes. Each node $v \in V$ has a unique* port number *assigned to each neighbor from the range $1, 2, \ldots, \deg(v)$ (where $\deg(v)$ denotes the* degree*—i.e., number of neighbors—of v). Nodes are modeled by finite state machines $T_v = (S, s_{v,0}, \Sigma, \Lambda, t, o)$, where the node-dependent initial state $s_{v,0}$ is referred to as the* input *of node v. The input and output alphabets are of the form $\Sigma = (\Sigma_0 \times \{1, \ldots, \deg(v)\}) \cup \Gamma$ and $\Lambda = \Sigma_0^{\deg(v)} \times \Omega$, where Σ_0 is the set of admissible messages.*[4] *Σ_0 and Ω contain the special symbol $\perp$, which we will use to indicate "no message" or "no output," respectively. An algorithm is executed based on* events*, where an* event at node $v \in V$ *consists of a node (1) starting to execute the algorithm, (2) receiving a local input from Γ, or (3) receiving a message from a neighbor; nodes start to locally execute the algorithm at the latest on receipt of the first message. Upon the i-th event at node v, v performs the following steps:*

1. *if the event is reception of a message m on port p, set $x := (m, p)$; if it was the reception of local input γ, set $x := \gamma$; otherwise set $(m, p) := (\perp, 1)$;*

[4] Again, we omit the resulting dependency of the state machine on $\deg(v)$ from the notation.

2. *update its state, i.e.,* $s_{v,i} := t(s_{v,i-1}, x)$*;*
3. *for* $p \in \{1, \ldots, \deg(v)\}$*, send message* $o(s_{v,i-1}, x)_p$ *to the neighbor whose port at* v *is* p*; and*
4. *output* $o(s_{v,i-1}, x)_{\deg(v)+1}$.

If after event i *the FSM of* v *reaches a state* s *satisfying that*

- $t(s_{v,i}, \cdot) = s_{v,i}$ *(no further state change possible),*
- $o(s_{v,i}, \cdot)_p = \perp$ *for* $p \in \{1, \ldots, \deg(v)\}$ *(no further messages can be sent), and*
- $o(s_{v,i}, \cdot)_{\deg(v)+1} = o_v$ *for some* $o_v \in \Omega$ *(no further output change),*

we say that v terminated after i steps with output o_v*. and call* s *a* terminal state.

Note that each sent message will be received, but we do not assume anything about when *this happens, except that messages are received after they are sent.*

Remark 6.13. *The fact that it is not specified when messages are received in the AMP model results in* non-deterministic *executions. In contrast to the SMP model, where the initial states fully determine the state of the system in all rounds, in AMP the order in which messages are received might affect how the node states evolve. We stress that "non-deterministic" does not automatically mean "random," let alone "uniformly random." Ideally, we want that our algorithms work correctly and efficiently for* all *possible executions in the AMP model!*

The AMP model has a suprising relationship with time. It matters only in so far as it determines the (total) order in which events happen. The model allows for a step, i.e., some node processing an event, to take a femtosecond or 100 years. It is also possible that several nodes take steps concurrently at the same time, so long as we maintain that all messages are in transit for a positive amount of time. However, we could marginally move these concurrent events in time so that they are not concurrent any more, without affecting the computations performed by the nodes or the messages they send. Thus, so long as the network does not interact with the "outside" world, in the AMP model the role of time can be astracted to a total order of events such that if event B is receiving a message sent on event A, then A occurred before B.

Another peculiarity is that the description of the AMP model assumes that nodes respond to events immediately. Thus, it may appear that nodes perform calculations instantaneously! This is not the case, however. We can account for the time required for local computations in response to an event in the transit time of messages sent in response to it. That is, if a node v sends a message to w in response to some event, then the amount of time elapsed between when the event occurs and when w receives the message—which we refer to as the

end-to-end *delay* of the message—includes both the time it takes v to perform its local computations as well as the actual delay from when v sends the message to when w receives the message.

Any algorithm in the AMP model can be executed in the synchronous model: if all nodes start the execution at time 0 and the delay of each message is exactly 1 time unit, then the corresponding execution in the AMP model mimics the respective execution in the SMP model. Thus, the AMP model is a generalization (or relaxation) of the SMP model in the sense that every execution of an SMP algorithm has a corresponding execution in the AMP model.

Thus far, our description of the AMP model does not allow us to measure the "speed" of an algorithm. Indeed, since we make no assumptions about timing, a node could take aribtrarily long to perform a single, simple operation. In order to give some quantitative measure of how time-efficient an algorithm is in the AMP model, we must make additional assumptions. In what follows, we assume that delay of *the slowest* message, the *maximum delay,* is one unit of time. That is, no more than one time unit lies between the event causing a message to be sent and the event of the message being received. The *complexity* of an algorithm is then the maximum number of time units elapsed for any execution of the algorithm in which the first node(s) locally start the execution at time 0.

Definition 6.14 (Asynchronous time complexity)**.** *Let $\mathcal{A}$ be an algorithm in the AMP model, and let $\mathcal{T}$ denote a task performed by $\mathcal{A}$. The* time complexity *of $\mathcal{A}$ completing $\mathcal{T}$ is defined to be*

$$\sup\{r_1 - r_0 \mid \mathcal{T} \text{ initiated at time } r_0,\ \text{completed at time } r_1\}, \tag{6.3}$$

where, for a fixed network G, the supremum is taken over all possible states of the individual nodes in round r_0 and all possible transit times of at most 1*. That is, the time complexity of $\mathcal{A}$ completing $\mathcal{T}$ on G is the maximum time that elapses between when $\mathcal{T}$ is initiated until it is completed, provided that messages are received at most* 1 *time unit after they were sent.*

Remark 6.15. *It is important to understand that the restriction to a maximum transit time of* 1 *is without loss of generality; it is a normalization that enables somewhat fair comparison between algorithms. For any execution in which the maximum message transit time during $[r_0, r_1]$ equals Δ, we can construct an execution with the "correct" maximum message transit time by letting events that occur at time t instead occur at time $r_0 + (t - r_0)/\Delta$. As this does neither change the order of events nor the nodes' response to them, the resulting execution has maximum transit time* 1 *during $[r_0, r_0 + (r_1 - r_0)/\Delta]$.*

In other words, in executions with maximum end-to-end delay Δ, tasks with asynchronous time complexity T take at most ΔT time.

Remark 6.16. *In general, it is beyond the control of the algorithm when local inputs arrive. In order for the above notion of time complexity to be meaningful, one needs that either the task description does not require nodes to wait for local inputs when performing a task (although they might very well trigger the task!) or make assumptions on their timely arrival.*

E6.6 Can you come up with meaningful asynchronous variants of the "simultaneous" restart problem? What is their time complexity? What else do they guarantee, and what not?

We conclude with the following remarks.

- The AMP model is extremely pessimistic about timing, as it assume no bounds whatsoever on computation time and how messages are in transit.
- This means its realistic in the sense that algorithms in this model can be readily executed by the physical hardware (assuming we made sure there are no faults!). However, the highly pessimistic assumptions on timing might render the result (too) inefficient in practice.
- The SMP and the AMP models are two extremes, so understanding them also helps understanding the range in between.

6.4 Simulating Synchrony

As we suggested above, the main conceptual advantage of the SMP model is that in the SMP model it is relatively easy to design algorithms and reason about their performance. Given the unrealistic assumptions made in the SMP model, however, one may wonder how useful such algorithms are in practice. When can an SMP algorithm be implemented in the more realistic AMP model?

In this section, we show the perhaps surprising result that *every* SMP algorithm can be transformed into an equivalent algorithm in the AMP model. The central idea is that of *simulation*: given any SMP algorithm, we can construct an AMP algorithm that simulates each round of the SMP algorithm in the AMP model.

The basic idea of the simulation is simple, and essentially boils down to simulating the round counter from Example 6.4. Each node v maintains a round count i corresponding to the next round of the SMP computation it wishes to simulate. The node v waits until all of its neighbors' round counters are at least $i-1$, which it learns from messages sent by its neighbors. In other words, whenever a node increases its counter to some value i, it sends out

a message indicating this to each of its neighbors, and it *waits for all* of its neighbors' corresponding messages before increasing its counter to value $i+1$. At this point, v has all of the information it needs to perform round $i+1$, so it performs the respective computations, sends messages to its neighbors, and waits to proceed to the next round of its computation. Below, we formalize this idea, and prove that it faithfully simulates the execution of any SMP algorithm.

We begin by formally defining what we mean by simulation. Intuitively, this simply means that the simulating algorithm collects all the information needed at each node to locally "run" the FSM of the simulated algorithm. This is a bit tricky to express, as the simulating algorithm might need to do a lot of bookkeeping and waiting for messages, etc., to ensure progress of the computation.

Definition 6.17 (Simulating SMP in AMP)**.** *AMP algorithm $\mathcal{A}$* simulates *SMP algorithm $\mathcal{B}$, if the following holds. Let $T_v^{\mathcal{A}} = (S^{\mathcal{A}}, s_{v,0}^{\mathcal{A}}, \Sigma^{\mathcal{A}}, \Lambda^{\mathcal{A}}, t^{\mathcal{A}}, o^{\mathcal{A}})$ and $T_v^{\mathcal{B}} = (S^{\mathcal{B}}, s_{v,0}^{\mathcal{B}}, \Sigma^{\mathcal{B}}, \Lambda^{\mathcal{B}}, t^{\mathcal{B}}, o^{\mathcal{B}})$ be the state machines of $\mathcal{A}$ and $\mathcal{B}$ at node $v \in V$, respectively. We require that $\Gamma^{\mathcal{A}} = \Gamma^{\mathcal{B}}$ and $\Omega^{\mathcal{A}} = (\Omega^{\mathcal{B}})^*$, i.e., $\Omega^{\mathcal{A}}$ is the set of words over alphabet $\Omega^{\mathcal{B}}$; the empty word ε indicates that the simulation did not progress in the current step of $\mathcal{A}$. Now, if for all $v \in V$*

- *$s_{v,0}^{\mathcal{A}} = s_{v,0}^{\mathcal{B}} = s_{v,0}$ for feasible inputs $s_{v,0}$ and*
- *the concatenation of local inputs to $\mathcal{A}$ at v equals the sequence $(\gamma_{v,i}^{\mathcal{B}})_{i \in \mathbb{N}_{>0}}$ of local inputs to $\mathcal{B}$,*

then concatenating the output words of $\mathcal{A}$ is required to produce the same word at all $v \in V$ as concatenating the output symbols of $\mathcal{B}$.

E6.7 Show that it is possible that $\mathcal{B}$ terminates at a node $v \in V$ without $\mathcal{A}$ doing the same when reaching the respective point in the simulation.

E6.8 Modify $\mathcal{B}$ such that we can infer from the output of $\mathcal{A}$ when $\mathcal{B}$ has terminated at $v \in V$. Can $\mathcal{A}$ now always terminate when reaching the point in the simulation when $\mathcal{B}$ does, or could there be a need to run $\mathcal{A}$ beyond this point?

Theorem 6.18. *Given any algorithm $\mathcal{B}$ in SMP, there is an algorithm $\mathcal{A}$ that simulates $\mathcal{B}$ in AMP. The asynchrononous time complexity of simulating the first T rounds of $\mathcal{B}$ is $t_0 + T$ in executions where all nodes $v \in V$ have woken up by time t_0 and for all $i \in \mathbb{N}_{>0}$ the event of v receiving $\gamma_{v,i}^{\mathcal{B}}$ happens by time $t_0 + i$. In graphs of diameter D, this implies an asynchronous time complexity of $D + T$ for simulating the first T rounds of $\mathcal{B}$ in such executions.*

Proof. Our strategy is for each node to perform the computations of $\mathcal{B}$ in the first round and send the respective messages. For subsequent rounds, nodes

Algorithm 2 Response of the simulation algorithm $\mathcal{A}$ to an event at node $v \in V$. The pseudocode does not explicitly keep track of the state updates of $\mathcal{A}$, collecting the output generated by reception of a message into a single word, or outputting the empty word ε if the simulation makes no local progress.

if v just woke up **then**
  $\text{round}_v := \text{input}_v := 1$
  $s^{\mathcal{B}}_{v,1} := t(s^{\mathcal{B}}_{v,0}, (\bot, \dots, \bot))$
  for $p \in \{1, \dots, \deg(v)\}$ **do**
    send $(o^{\mathcal{B}}(s_{v,0}, (\bot, \dots, \bot))_p, 1)$ to port p
  end for
  output $o^{\mathcal{B}}(s_{v,0}, (\bot, \dots, \bot))_{\deg(v)+1}$
end if
if v received $(m^{\mathcal{B}}, i)$ on port p **then**
  store $(m^{\mathcal{B}}, i, p)$
end if
if v received local input γ **then**
  store (γ, input_v)
  $\text{input}_v := \text{input}_v + 1$
end if
while v stores (γ, round_v) and $(m^{\mathcal{B}}_p, \text{round}_v, p)$ for all $p \in \{1, \dots, \deg(v)\}$ **do**
  $\text{round}_v := \text{round}_v + 1$
  $s^{\mathcal{B}}_{v,\text{round}_v} := t^{\mathcal{B}}(s^{\mathcal{B}}_{v,\text{round}_v-1}, (m^{\mathcal{B}}_1, \dots, m^{\mathcal{B}}_{\deg(v)}, \gamma))$
  for $p \in \{1, \dots, \deg(v)\}$ **do**
    send $(o^{\mathcal{B}}(s^{\mathcal{B}}_{v,\text{round}_v-1}, (m^{\mathcal{B}}_1, \dots, m^{\mathcal{B}}_{\deg(v)}))_p, \text{round}_v)$ to port p
  end for
  output $o^{\mathcal{B}}(s^{\mathcal{B}}_{v,\text{round}_v-1}, (m^{\mathcal{B}}_1, \dots, m^{\mathcal{B}}_{\deg(v)}))_{\deg(v)+1}$
end while

wait until they have received all messages from prior rounds and the next input symbol. To ensure that this can be locally verified, nodes label their messages with the round number and will send "empty" messages (i.e., a $(\bot, i)$ messsage for round i) whenever $\mathcal{B}$ does not send a message to a neighbor in a given round. The pseudocode for this approach is given in Algorithm 2.

First, we claim that there is a time t_0 by which all nodes have woken up. To see this, recall that there is some node $v \in V$ waking up at time 0. As nodes send messages when waking up, all nodes in distance 1 of v (i.e., its neighbors) wake up by the time these messages arrive. They send messages, too, waking up nodes in distance 2 from v, and so on. As G is connected, we see that

all nodes in G are woken up by chains of messages of length at most D, the diameter of G. Note that in executions with maximum delay 1, this entails that $t_0 \leq D$. Hence, it remains to show that $\mathcal{A}$ correctly simulates the first T rounds of $\mathcal{B}$ by time $t_0 + T$ in executions with maximum delay 1 in which each $v \in V$ receives local input $\gamma_{v,i}^{\mathcal{B}}$ by time $t_0 + i$.

We prove this statement by induction on T, where the induction hypothesis is that the state updates $\mathcal{A}$ computes, the messages it sends, and the output it generates for rounds $r \leq T$ are correct and timely. That is, for all $i \leq r$, $v \in V$, and $p \in \{1, \dots, \deg(v)\}$,

- $\mathcal{A}$ sets $s_{v,i+1}^{\mathcal{B}}$ to the same value as $\mathcal{B}$ by time $t_0 + i$,
- sends message $(m, i+1)$ to port p by time $t_0 + i$, where m is the message $\mathcal{B}$ sends to port p in round $i+1$,
- receives message $(m, i+1)$ on port p by time $t_0 + i + 1$, where m is the message $\mathcal{B}$ receives on port p in round $i+1$, and
- outputs by time $t_0 + i$ the symbol that $\mathcal{B}$ outputs in round $i+1$, but not before outputting the respective symbol for round i (if $i > 0$).

We anchor the induction at $r = 0$ by observing that (i) $s_{v,0}^{\mathcal{B}}$ is provided to $\mathcal{A}$ as input, (ii) it computes the state, messages to send, and output that $\mathcal{B}$ would for round 1 immediately (i.e., by time t_0), and (iii) the respective messages are received by time $t_0 + 1$.

For the induction step from $r-1$ to r, observe that by the induction hypothesis, we only need to check the respective statements for index $i = r$. From the hypothesis for $i = r - 1$, we have that all round r messages are received by time $t_0 + r$. By the prerequisites of the theorem, $v \in V$ receives input $\gamma_{v,r}$ by time $t_0 + r$, and by Definition 6.17 after receiving the previous input symbols; as the algorithm keeps count of the number of input symbols it received, it stored $(\gamma_{v,r}, r)$ by time $t_0 + r$. Hence, the while loop lets v compute $s_{v,r+1}^{\mathcal{B}}$, the messages it sends in round $r+1$ of $\mathcal{B}$, and the output of $\mathcal{B}$ in round $r+1$ by this time. As the sent messages are received by time $t_0 + r + 1$ and the output for round $r+1$ is generated after the one for round r (as the variable round_v is only ever increased), this completes the induction step and thus the proof. □

E6.9 The simulation uses counter values i that grow indefinitely and stores the entire history of the computation. Make it work with constant-sized modulo counters, messages of $\lceil \log \Sigma_0 \rceil + 1$ bits and $2 \deg(v)$ message buffers each holding $\lceil \log \Sigma_0 \rceil$ bits (but no limit on the memory dedicated to input events)! Hint: Show that for each $\{v, w\} \in E$, $|\text{round}_v - \text{round}_w| \leq 1$ at all times. Use this to prove that the simulation of round i at v is completed before any message $(m, i+2)$ can be received by v.

Corollary 6.19. *If there are no local inputs, i.e.,* $\Gamma = \{\gamma\}$ *for a dummy symbol* γ*, Theorem 6.18 holds without any conditions on input events, where the simulation algorithm* $\mathcal{A}$ *uses messages of* $\lceil \log \Sigma_0 \rceil + 1$ *bits and at most* $2 \deg(v) \lceil \log \Sigma_0 \rceil + 1$ *bits of additional memory at* v *than* $\mathcal{B}$*.*

Proof. Instead of receiving and storing input symbols labeled according to their order of arrival, the simulation algorithm simply ignores all input events and always plugs in γ when calling t or o. It needs on bit of memory to distinguish between odd and even rounds, i.e., it only stores $\text{round}_v \bmod i$, and also uses only this value when sending messages. The memory requirements then follow by using different sets of message buffers for odd and even rounds, which can safely be reused once the simulation of the respective round is complete. □

Remark 6.20. *In general, if the simulated algorithm has non-trivial inputs, the memory overhead of the simulating algorithm might be arbitrarily large. For example, we could require that* $\mathcal{A}$ *in round* i *at node* v *outputs* 0 *if any of its neighbors received an input of* 0 *in the previous round, but* 1 *otherwise. This clearly can be implemented easily with bounded memory in the synchronous setting. However, the simulation algorithm* $\mathcal{A}$ *might suffer from an extremely slow node. While messages from* w *do not arrive, but input events occur at the other neighbors of* v*, the system state must maintain information on these input events. Regardless of whether this is done by storing it at nodes or "storing" it in messages that bounce back and forth between nodes, we cannot bound the amount of hardware required if there are not guarantees on the timing of input events whatsoever. This is an example where worst-case assumptions on timing become impractical, and more restrictive assumptions need to be made—and justified.*

E6.10 What do you get when applying Theorem 6.18 to the algorithm solving the synchronous restart problem from Section 6.2.1.1? Can you see some practical utility of the result?

Bibliography