9 A Limit on Fault Tolerance

Chapter Contents

Learning Goals

In this chapter, we study how *Byzantine faults*, i.e., components that may behave in any possible way due to failure or corruption, affect whether synchronization can be achieved among the correct nodes of a network [5]. We consider the same model as in the previous chapters – nodes have local clock sources of bounded drift and messages have bounded delay – and in addition, to simplify the discussions, we focus in this chapter on fully connected networks. We prove that synchronization cannot be achieved in n-node networks with $f \geq \frac{n}{3}$ faults. This bound is tight: we present a simple variant of the algorithm by Srikanth and Toueg [6] that solves the *pulse synchronization* task, in which the correct nodes generate synchronized (clock) pulses in a regular fashion. As will be discussed, pulse synchronization is essentially the same as clock synchronization, to which the same bounds on the number of sustainable Byzantine faults apply. We choose this variant of the clock synchronization problem here, as it naturally lends itself to simulating lock-step execution. We discuss an implementation of

the presented pulse synchronization algorithm and use it to simulate lock-step execution in detail.

The possible primary learning objectives of this chapter are:

- The concept of Byzantine faults and its relevance (Section 9.2).
- Relation between pulse and clock synchronization (Section 9.3).
- Limits on the number of Byzantine faults that can be tolerated (Sections 9.2, 9.4, and 9.5).
- Simulation of lock-step execution in face of Byzantine faults (Sections 9.3.2, 9.5, and 9.6).

9.1 Overview

As shown in Chapter 7, the possibility of a crash fault renders the simulation of SMP in AMP given in Chapter 6 impossible. This prompted us to equip nodes with local clock sources of bounded drift and ensure that communication delays satisfy known bounds. However, the synchronization techniques from Chapter 7 are inherently limited to handling benign faults only, as all information is assumed to be correct.

E9.1 Recall the algorithms from Chapter 7 and think of ways they could break due to non-crash faults.

It is not hard to argue that assuming crash faults is too optimistic. But which fault model to use?

E9.2 Consider modes of failure against which one might want to protect the system.

E9.3 Think about the pros and cons of deriving fault models from studying and taking measurements of physical implementations.

Preparing for typical faults is insufficient for large or high-reliability systems, as this is likely to result in algorithms that may fail due to a single "atypical" fault. In addition, only substantial restrictions of the fault model that can be exploited by algorithms are going to be useful. These considerations motivate to study the extreme case of Byzantine faults, where no restrictions are put on the behavior of faulty nodes (except for an inability to cause other components to also violate their specification). This fault model and the underlying rationale are discussed in more detail in Section 9.2.

Definition 9.1 (Byzantine Faults)**.** *A* Byzantine *faulty node is a node that may behave arbitrarily. That is, such a node does not need to follow any algorithm prescribed by the system designer. An algorithm is* resilient *to f Byzantine faults if its performance guarantees hold for any execution in which there are at most f Byzantine faulty nodes. In the following, for a network $G = (V, E)$ and a set $F \subseteq V$ of faulty nodes, we denote by $V_g = V \setminus F$ the set of correct nodes.*

Unsurprisingly, there are limits on the amount of Byzantine faults that can be sustained. The reader should find it intuitive that a node with a majority of faulty neighbors cannot (reliably) stay synchronized to the other correct nodes in the system.

Theorem 9.2. *Fix any network of $n \geq 3$ nodes with a node of degree at most $2f$. Then, for any clock synchronization algorithm with non-trivial progress guar-*

antee (Definition 7.10), there is an execution on this network with unbounded skew.

E9.4 Spend a few minutes to think about how this theorem might be proven.

Recall that our goal in this chapter is to understand how many Byzantine faults can be sustained, which we want to express as a function of n. The above theorem establishes that, in particular, $f < \frac{n}{2}$, and that $f \in \Theta(n)$ necessitates node degrees in $\Theta(n)$. With this in mind, we restrict our attention to fully connected networks throughout this and some subsequent chapters. However, note that it is meaningful to study fault-tolerance in networks with smaller degrees, as low-degree topologies scale much better with n. Especially when faulty nodes are not chosen adversarially, but rather distributed at random, a fairly large fraction of faulty nodes can be sustained even with small degrees.

In this chapter, we study a variant of clock synchronization known as pulse synchronization.

Definition 9.5. *[Pulse Synchronization] In* pulse synchronization*, for each $i \in \mathbb{N}$, every (non-faulty) node $v \in V_g$ generates* pulse *i exactly once. Let $p_{v,i}$ denote the time when v generates the i-th pulse. We require that there are $\mathcal{S}, P_{\min}, P_{\max} \in \mathbb{R}^+$ satisfying*

1. $\sup_{i\in\mathbb{N},v,w\in V_g}\{|p_{v,i} - p_{w,i}|\} = \mathcal{S}$ *(skew).*
2. $\inf_{i\in\mathbb{N}}\{\min_{v\in V_g}\{p_{v,i+1}\} - \max_{v\in V_g}\{p_{v,i}\}\} \geq P_{\min}$ *(minimum period).*
3. $\sup_{i\in\mathbb{N}}\{\max_{v\in V_g}\{p_{v,i+1}\} - \min_{v\in V_g}\{p_{v,i}\}\} \leq P_{\max}$ *(maximum period).*

This task, which is discussed in more detail in Section 9.3, requires correct nodes to generate synchronized local events, *pulses*, with small skew and lower and upper bounds on their frequency. It is tailored specifically to making simulation of lock-step execution easy.

Theorem 9.7. *Suppose that nodes $v \in V_g$ run Algorithm 9 alongside a pulse synchronization algorithm and that the constraints given by Equations (9.1) to (9.4) are satisfied. Then the compound algorithm simulates lock-step execution of the algorithm described by the subroutines* $\mathrm{state}(s, m_{\mathrm{in}})$ *and* $\mathrm{msg}(s, m_{\mathrm{in}}, i)$ *(plus the initial states) at a rate of one round per pulse, i.e., at least one round per $P_{\max}$ time.*

Pulse synchronization appears to demand less than clock synchronization problem: pulse synchronization imposes constraints only on the discrete pulse events, whereas clock synchronization has similar requirements on the continuous logical clocks at all times. However, as we will show, the two tasks are essentially the same. Concretely, we will show that an algorithm for one can

be used to derive a solution for the other without additional communication or significant computational overhead.

Theorem 9.8. *Assume that for $T > \mathcal{G}$, the nodes in V_g run a clock synchronization algorithm with global skew $\mathcal{G}$, minimum clock rate 1, and maximum clock rate β. By adding only local computations to the algorithm, we can solve pulse synchronization with $\mathcal{S} = \mathcal{G}$, $P_{\min} = \frac{T-\mathcal{G}}{\beta}$, and $P_{\max} = T + \mathcal{G}$.*

In the above theorem, the additional local computation is limited to ensuring that for each $i \in \mathbb{N}$ the node generates a pluse at logical times iT. More is required for the opposite direction. Roughly speaking, we increase the logical clock value by $\vartheta P_{\max}$ per pulse, using the hardware clock as reference in between pulses. However, because pulses may be as close in time as $P_{\min}$, the logical clock value may be up to $\vartheta P_{\max} - P_{\min}$ too small at a pulse. This is handled by increasing the logical clock faster than the hardware clock for $P_{\min}$ local time, spreading out the increase as much as possible while guaranteeing that the deficit is removed before the next pulse occurs.

Theorem 9.11. *Suppose the nodes in V_g run a pulse synchronization algorithm with skew $\mathcal{S}$, minimum period $P_{\min}$, and maximum period $P_{\max}$. Furthermore, assume that node $v \in V_g$ generates its first pulse at real time $p_{v,0} \in [-\mathcal{S}, 0]$. By adding only local computations to the algorithm, we can solve clock synchronization with global skew $(\vartheta - 1)P_{\max} + \beta\mathcal{S}$, minimum clock rate 1, and maximum clock rate $\beta := \frac{\vartheta^2 P_{\max}}{P_{\min}}$.*

After establishing the above equivalence, the chapter focuses on determining for which number f of faulty nodes the two problems can be solved. First, we prove that no algorithm can solve pulse synchronization in the presence of $f \geq \frac{n}{3}$ Byzantine faults. This is shown in Section 9.4 by invoking a more elaborate indistinguishability argument. For an arbitrary given algorithm, we assume towards contradiction that it is correct. We then construct a sequence of executions $(\mathcal{E}_i)_{i \in \mathbb{N}_0}$ in which pulses are generated faster in $\mathcal{E}_{i+1}$ than in $\mathcal{E}_i$. However, for sufficiently large i this implies the contradiction that any possible upper bound on the frequency at which pulses are generated must be violated.

Theorem 9.13. *Pulse synchronization is impossible if $3 \leq n \leq 3f$.*

E9.5 Figure out why this impossibility does not hold for $n = 2$.

This bound is not coincidental: it can be precisely matched. We show this by presenting an algorithm based on what we refer to as *propose-pull voting,* which appears in many variants in synchronization primitives resilient to Byzantine faults.The technique was introduced by Srikanth and Toueg [6], who solve the

pulse synchronization task formalized in Section 9.3. The idea is that, after being initialized to some starting state, nodes will "propose" to generate a pulse after some local time has passed. When a node has received respective messages for pulse 1 from $n - f$ different nodes (counting itself as well), it generates pulse 1. However, alone this rule risks that some nodes are "left behind," as some of these $n - f$ supporting nodes may have been faulty and not sent respective messages to other nodes. That is where the "pulling" comes in: receiving messages for pulse 1 from $n - 2f \geq f + 1$ different nodes will prompt a node to send messages for pulse 1, too. By using pulse i to "re-initialize" the system for pulse $i + 1$, this mechanism can be used to inductively generate pulses for each $i \in \mathbb{N}$ with suitable period and skew bounds.

E9.6 Argue that if *some* correct nodes generates pulse i, *all* of the at least $n - f$ correct nodes generate pulse i within $2d$ time, where d is the maximum end-to-end delay.

The frequency at which pulses are generated can be controlled by adjusting the local time span between a node generating pulse i and proposing pulse $i + 1$ on its own.

Theorem 9.17. *Suppose $3f < n$, $H_v(0) \in [0, H_0)$ for all $v \in V$ and some known $H_0 \in \mathbb{R}^+$, and choose any $T \geq 3\vartheta d$. Then we can solve the pulse synchronization problem with $\mathcal{S} = 2d$, $P_{\min} = T$, and $P_{\max} = \vartheta T + (5 + 2(\vartheta - 1))d$, where each node generates its first pulse by time $H_0 + (\vartheta - 1)T + (3 + 2(\vartheta - 1))d$.*

Our variant of the Srikanth-Toueg algorithm is phrased in terms of a simple state machine each node implements. This paves the way to a hardware-level implementation discussed in Section 9.6. **??** derives an implementation that can simulate lock-step execution of an arbitrary distributed algorithm.

9.2 Byzantine Faults

A *Byzantine* faulty node (process, link, etc.) is one that may misbehave in an arbitrary way. This includes behavior that appears to be consistent with that of a correct node for a long time, only to violate the protocol at the worst possible time, as well as collusion among faulty nodes, i.e., "teamwork" to bring down the system as a whole. The term was coined by Pease, Shostak, and Lamport [5], alluding to the idea that in the final days of the Byzantine empire, corruption was so widespread that the generals of an army would have to use decision procedures that succeed even in the presence of (a minority of) conspiring traitors.

We must stress that our point of view throughout this book is *not* that we assume that there is an adversarial mastermind trying to orchestrate the downfall of our computing systems. While this is a suitable mindset for making a system secure, our main motivation is different: we want that our systems withstand as many fault patterns and unexpected faults as possible.

E9.7 Think about advantages this rather harsh fault model might provide even without an organized attack on the system.

The practical benefits of this strategy are several:

Coverage: Allowing any behavior of a component removes the need to develop a more detailed fault model and/or reliable components. Being prepared for anything means to be, in particular, prepared for the faults that occur in practice, without the need to figure out what these are in advance. This advantage should not be underestimated, as in large or complex systems it can be a great challenge to track down the cause of a system failure!

Testing: A related advantage is to reduce the burden of verifying that an assumed failure model is realistic. While developing a suitable fault model may succeed based on explorative observations and intuition, verification may require extensive experiments. Especially when high reliability is required, testing may become prohibitively expensive. In order to verify a mean time between failures of years, running an experiment takes too long. This implies that any practical test must itself rely on further model assumptions, which get more complex – and thus possibly less reliable – if the assumed fault model is more complex. To make things worse, testing may be an ongoing effort, as the possibility of changes in conditions for production or operation must be considered. For Byzantine faults, all these efforts are limited to ensuring a sufficiently small overall failure rate and that faults are *contained,* which is discussed in Section 9.2.1.

Scalability: Increasing system size or lifetime increases the likelihood of rare faults. A failure rate of 10^{-10} may appear small, but is clearly insufficient when it specifies the probability of failure per pulse of a system clock: at 3 GHz, this equates failure within seconds! Similarly, the probability that an individual transistor is not working after production of a chip is very small, yet we can be essentially certain that *some* of the huge number of transistors on a chip are faulty.

Reusability: All of the above challenges have to be overcome for every generation of a chip. Having a catch-all solution covering most conveivable faults can reduce time and effort for adapting solutions to the next generation – or even transferring them to a completely different setting – greatly.

The elephant in the room is, of course, that these considerable benefits of the fault model stand against its obvious downside: The fault model may be overly pessimistic, resulting in waste of resources or impossibility results that have an, at best, tenuous connection to reality. Any designer is well-advised to bear this issue in mind, and carefully contemplate which fault model to use. Accordingly, it is crucial to understand the cost at which the above advantages come, so we can make an informed decision regarding when to opt for more restrictive fault models. In reality, such considerations are likely to result in Byzantine fault-tolerance – and the redundancy it entails – being applied to critical subsystems only.

At first glance, it may be surprising that studying Byzantine faults is worthwhile even in cases where, ultimately, we opt for a more restrictive fault model. The reason is that lower bounds and impossibility results demonstrating the limitations imposed by the fault model frequently also reveal what exactly prevents us from achieving better results. Based on such insights, we can find minimal (and, accordingly, more likely still realistic) restrictions to the fault model enabling solutions with better guarantees. Several examples for this can be extracted from this chapter. As already mentioned, we will see that restricting the number f of Byzantine faults in an n-node system to $\lceil \frac{n}{3} \rceil - 1$, i.e., $3f < n$, suffices to solve pulse synchronization. Second and less immediate is an insight that can be gleaned from the following theorem, which shows that tolerating f Byzantine faults requires node degrees of at least $2f + 1$.

Theorem 9.2. *Fix any network of $n \geq 3$ nodes with a node of degree at most $2f$. Then, for any clock synchronization algorithm with non-trivial progress guarantee (Definition 7.10), there is an execution on this network with unbounded skew.*

Proof. (This proof will be discussed in the lecture and added to the chapter at a later point.) □

While this theorem shows that we need large degrees to overcome Byzantine faults if the *selection* of faulty nodes is done in a worst-case fashion, this dramatically changes if faulty nodes are chosen (uniformly) at random. In **??** we follow this idea, leading to a fault-tolerant low-degree clock distribution network.

9.2.1 Fault Containment Regions

The Byzantine fault model expects that we preserve correct operation of the system at (all) correct nodes. As this is not possible if too many nodes (links, components, etc.) fail, we should be wary of single events causing multiple

faults. As an extreme example, consider a "fault-tolerant" system-on-chip that is powered from a single supply without any backup. Our abstract model is unlikely to explicitly contain the power supply. Yet, if the power supply fails, all of the nodes in the system "fail" from the perspective of any algorithm we might come up with. It may seem obvious that a careful designer needs to make sure that the power supply is sufficiently robust to fail so rarely as to meet the design goals or that fallback mechanisms (emergency power supply, batteries, etc.) must be in place.

What may seem trivial in the above example may be harder to catch in other cases. For instance, in the node-centric perspective we take throughout this book, a failing communication link is interpreted as a failure of either the sender or the receiver. However, if communication is via a bus, a failure of the bus again means that – from the perspective of our model – all nodes become faulty. Naturally, a careful designer will have avoided using a bus and may even have routed the communication wires carefully to avoid or minimize crossings. Consequently, for any (localized) error the caused communication faults can be mapped to a single node, meaning that the resilience provided by a Byzantine fault-tolerant algorithm is meaningful for this kind of error. Unfortunately, this foresight may be rendered ineffective if a different designer later takes the solution and maps it to a different system with a bus-based communication infrastructure, not realizing that this changes which fault events the resilience guarantee of the fault model covers for the new system. Another case that causes a violation of the resilience guarantee is having a new team that decides to make the system more cost efficient by reducing what seems to be wasteful resources.

In order to avoid such issues, it is advisable to formalize the avoidance of a "single point of failure" showcased by the examples above. To this end, Kopetz [4] coined the term *fault containment regions,* attributing the concept to Hopkins et al. [3].

Definition 9.3 (Fault Containment Regions)**.** *A fault containment region is defined as a set of subsystems that are potentially affected by a single fault. This definition implicitly assumes a defined set of possible faults, which determines what is to be considered a fault containment region.*

E9.8 Consider a system you know well and think about how it would have to be reorganized into redundant fault-containment regions to allow for fault-tolerance. Which subsystems should you focus on first to increase reliability?

E9.9 Could you use less reliable components to (partially) offset the cost of adding redundancy?

E9.10 If you don't know the answers to these questions, what needs to be done to find out?

As these exercises and the preceding discussion illustrates, how meaningful Byzantine fault-tolerance is strongly depends on how well we can prevent single (likely) sources of faults from preventing a large(r) fraction of the system. Throughout this book, we will assume that nodes and their associated subsystems are fault-containment regions, but the reader should be painfully aware that this is only useful so long as this is close enough to reality.

9.3 Relation of Pulse Synchronization to other Tasks

Pulse and clock synchronization are, roughly speaking, the same task. We show this by proving that we can solve each problem based on a solution to the other. Note that in the context of Byzantine faults, any guarantees are limited to the set of correct nodes, so we first need to adjust the previous definition of a clock synchronization algorithm, Definition 7.2.

Definition 9.4 (Clock Synchronization with Byzantine Faults)**.** *In the* clock synchronization *task, each correct node $v \in V_g$ provides a subroutine for computing a* logical clock $L_v \colon \mathbb{R}_0^+ \to \mathbb{R}_0^+$*. The subroutine must be executed locally without waiting for other operations like external communication. An algorithm has*

- global skew $\mathcal{G}$*, if* $\max_{v,w \in V_g} \{|L_v(t) - L_w(t)|\} \leq \mathcal{G}$*...*
- minimum rate α*, if* $\frac{dL_v}{dt}(t) \geq \alpha$*...*
- maximum rate β*, if* $\frac{dL_v}{dt}(t) \leq \beta$*...*

... for all $t \in \mathbb{R}_0^+$ and $v \in V_g$.

Note that for a clock synchronization algorithm with minimum rate $\alpha > 0$, dividing all L_v by α results in a clock synchronization algorithm with minimum rate 1 and maximum rate β/α, where the skew bounds also scale by $1/\alpha$. For simplicity, we will hence fix $\alpha = 1$ throughout this chapter.

9.3.1 Definition and Basic Observations

Before dwelling into the simulation arguments of this section, we elaborate on the definition of pulse synchronization and prove a claim that will be useful for these simulation arguments.

Definition 9.5. *[Pulse Synchronization] In* pulse synchronization*, for each $i \in \mathbb{N}$, every (non-faulty) node $v \in V_g$ generates* pulse *i exactly once. Let $p_{v,i}$ denote the time when v generates the i-th pulse. We require that there are $\mathcal{S}, P_{\min}, P_{\max} \in \mathbb{R}^+$ satisfying*

1. $\sup_{i\in\mathbb{N},v,w\in V_g}\{|p_{v,i}-p_{w,i}|\}=\mathcal{S}$ *(skew).*
2. $\inf_{i\in\mathbb{N}}\{\min_{v\in V_g}\{p_{v,i+1}\}-\max_{v\in V_g}\{p_{v,i}\}\}\geq P_{\min}$ *(minimum period).*
3. $\sup_{i\in\mathbb{N}}\{\max_{v\in V_g}\{p_{v,i+1}\}-\min_{v\in V_g}\{p_{v,i}\}\}\leq P_{\max}$ *(maximum period).*

In this definition, the first condition ensures that corresponding pulses at different nodes are closely aligned in time. This condition is closely related to the global skew, although it is expressed in terms of the real time difference between pulse events rather than the logical time difference at a given real time. These notions are connected via lower an upper bounds on clock rates. The second and third condition bound the minimum and maximum real time, respectively, between consecutive pulses of any pair of nodes.

We first take note of some general constraints on these values.

Lemma 9.6. *Any pulse synchronization algorithm must satisfy that*

1. $P_{\max}-P_{\min}\geq\mathcal{S}$ *and*
2. $P_{\max}\geq\vartheta P_{\min}$.

Proof. For the first statement, we bound

Definition 9.5
$$\begin{aligned}\mathcal{S}&=\sup_{i\in\mathbb{N},v,w\in V_g}\{|p_{v,i}-p_{w,i}|\}\\&=\sup_{i\in\mathbb{N}}\{\max_{v\in V_g}\{p_{v,i}\}-\min_{v\in V_g}\{p_{v,i}\}\}\\\text{adding 0}\qquad&=\sup_{i\in\mathbb{N}}\{\max_{v\in V_g}\{p_{v,i+1}\}-\min_{v\in V_g}\{p_{v,i}\}-(\max_{v\in V_g}\{p_{v,i+1}\}-\max_{v\in V_g}\{p_{v,i}\})\}\\\max\geq\min\qquad&\leq\sup_{i\in\mathbb{N}}\left\{\max_{v\in V_g}\{p_{v,i+1}\}-\min_{v\in V_g}\{p_{v,i}\}-(\min_{v\in V_g}\{p_{v,i+1}\}-\max_{v\in V_g}\{p_{v,i}\})\right\}\\&\leq\sup_{i\in\mathbb{N}}\left\{\max_{v\in V_g}\{p_{v,i+1}\}-\min_{v\in V_g}\{p_{v,i}\}-\inf_{i\in\mathbb{N}}\left\{\min_{v\in V_g}\{p_{v,i+1}\}-\max_{v\in V_g}\{p_{v,i}\}\right\}\right\}\\&=\sup_{i\in\mathbb{N}}\left\{\max_{v\in V_g}\{p_{v,i+1}\}-\min_{v\in V_g}\{p_{v,i}\}\right\}-\inf_{i\in\mathbb{N}}\left\{\min_{v\in V_g}\{p_{v,i+1}\}-\max_{v\in V_g}\{p_{v,i}\}\right\}\\\text{Definition 9.5}\qquad&\leq P_{\max}-P_{\min}.\end{aligned}$$

For the second statement, consider two executions of the algorithm without faulty nodes. In the first execution, all delays are d and all hardware clock rates are 1 at all times. In the second execution, all delays are $\frac{d}{\vartheta}$ and all hardware clock rates are ϑ at all times. By induction over the times at which messages are sent and received, we see that the two executions are indistinguishable. Accordingly, all pulse events occur at the same local times in both executions. As hardware clock speeds differ by factor ϑ at all times in both executions, the second claim of the lemma follows. □

9.3.2 Simulating Lock-step Execution

In previous chapters, we have used synchronizers (Chapter 6) and clock synchronization algorithms (Chapter 7) to simulate synchronous executions. In this chapter, we do so based on a pulse synchronization algorithm. The approach is very similar to that for a clock synchronization algorithm, but instead of using the logical clock, we rely on pulses and the hardware clock to execute steps at the correct time. In contrast to Chapter 7, where the respective discussion remained abstract, in this chapter we will go into the details of an implementation of simulating a synchronous execution. In particular this requires us to simulate, i.e., implement, a synchronous communicating state machines. Accounting for this, in this subsection we slightly refine the model, in order to better capture the precise timing behavior needed to achieve the best possible performance.

Following the notation of Section 2.2, each node $v \in V$ runs a local state machine with state space S. Nodes operate in discrete common rounds where:

1. They receive messages from the previous round.
2. They then update their state via a local state machine's transition function $t : S \times \Sigma \to S$, where the input alphabet Σ is the set of all messages combinations.
3. They send a message to each node $u \in V$ that they have an outgoing link to. The message to be sent to u is determined by the local state machine's output function $o_u : S \to \Lambda$ for node u. The output alphabet Λ is the set of messages. While we favored Moore state machines in Section 2.2 for the sake of handling unstable inputs, we will also use the more compact Measly representation sometimes in the the following. In this case, $o_u : S \times \Sigma \to \Lambda$ depends on the combination of received messages.

We will next discuss the implementation of synchronous communicating state machines, and thus lock-step executions, given that nodes have access to a pulse synchronization algorithm.

In the following fix a node $v \in V$. Denote by $\mathrm{state}(s, m_{\mathrm{in}})$ a subroutine that maps a state $s \in S$ (from the previous round) and an array $m_{\mathrm{in}} \in \Sigma$ of $\delta_{\mathrm{in}}(v)$ messages (received on v's incoming ports in the previous round) to a new state (the one attained in the current round). Similarly, denote by $\mathrm{msg}(s, m_{\mathrm{in}}, i)$ a subroutine whose first two arguments are the same as for state plus an additional output port number $i \in \{1, \dots, \delta_{\mathrm{out}}(v)\}$ and that returns a message (the one to be sent on outgoing port i in the current round). These subroutines are given by the synchronous algorithm to be simulated (cf. Definition 6.17). Moreover,

the simulated algorithm provides for each node an initial state, and we assume that the elements of the array m_{in} are initialized to some default value $\perp$.

We introduce three pairs of parameters:

- $d_{\text{min,send}}$ and $d_{\text{max,send}}$ denote the minimum and maximum communication delay for sending a message to a neighbor, respectively (*without* any subsequent computations!)
- $d_{\text{min,state}}$ and $d_{\text{max,state}}$ denote the minimum and maximum computational delay, respectively, for a state update, i.e., the duration between starting to compute $\text{state}(s, m_{\text{in}})$ and the outcome propagating to the registers holding the state (including the time to latch).
- $d_{\text{min,msg}}$ and $d_{\text{max,msg}}$ denote the minimum and maximum computational delay, respectively, for computing a message, i.e., the duration between starting to compute $\text{msg}(s, m_{\text{in}}, i)$ and the outcome propagating to the registers holding the state (including the time to latch).

Note that one could refine the timing analysis even further, but our goal here is to maintain a good balance between performance and conceptual clarity.

Algorithm 9 provides pseudocode describing the actions that are taken to simulate a round. Figure 9.1 shows the local timing behavior produced by this code, provided the following constraints are satisfied:

local time offset for sending
$$\tau := \frac{P_{\text{min}}}{\vartheta} - \hat{\mathcal{S}} - d_{\text{max,send}} \tag{9.1}$$

compute state before receiving
$$d_{\text{max,state}} \leq \frac{\tau - \hat{\mathcal{S}}}{\vartheta^2} + \frac{d_{\text{min,send}}}{\vartheta} + d_{\text{min,state}} \tag{9.2}$$

compute mess. before receiving
$$d_{\text{max,msg}} \leq \frac{\tau - \hat{\mathcal{S}}}{\vartheta^2} + \frac{d_{\text{min,send}}}{\vartheta} + d_{\text{min,msg}} \tag{9.3}$$

compute mess. before sending
$$d_{\text{max,msg}} \leq \frac{\tau}{\vartheta} \tag{9.4}$$

Theorem 9.7. *Suppose that nodes $v \in V_g$ run Algorithm 9 alongside a pulse synchronization algorithm and that the constraints given by Equations (9.1) to (9.4) are satisfied. Then the compound algorithm simulates lock-step execution of the algorithm described by the subroutines* $\text{state}(s, m_{\text{in}})$ *and* $\text{msg}(s, m_{\text{in}}, i)$ *(plus the initial states) at a rate of one round per pulse, i.e., at least one round per P_{max} time.*

Proof. (This proof will be added later.) □

9.3.3 From Clock Synchronization to Pulse Synchronization

First, we prove that a clock synchronization with bounded clock rates can be readily used to solve pulse synchronization, by generating pulses at predefined

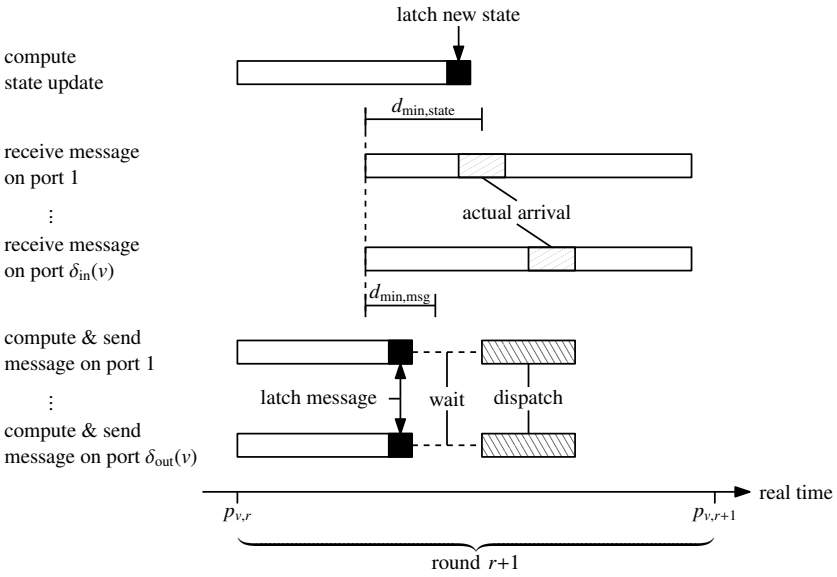

Figure 9.1
Illustration of the local timing behavior of Algorithm 9. Round $r+1$ at node $v \in V_g$ starts with pulse $p_{v,r}$. In parallel, the node computes its state update and the messages it sends in round $r+1$, based on its state and the incoming messages of round r. Note that the variable s holding the state and the array m_{in} holding the incoming messages are used by multiple threads, which requires careful timing to ensure correct ordering of accesses. Concretely, computations in round $r+1$ should be based on the messages received and the state computed in round r. We exploit that propagation through computational logic has a minimum delay, which is denoted by $d_{\min,\text{state}}$ and $d_{\min,\text{msg}}$ for the computation of the new state and outgoing messages, respectively. This is the reason why the node is ready for receiving messages only a while after the start of the round. Note also that there is a deliberate delay between latching the outgoing messages and actually dispatching them, accounting for the fact that the receiving nodes are not ready to receive messages earlier in the round. Finally, we stress that the bars indicate worst-case time bounds for the duration of the individual steps (in terms of local time). For instance, as indicated, the actual reception of a message make take only a small fraction of the alloted interval. Moreover, as v uses its local clock to measure these times and pulse durations may vary, up to $P_{\max} - \frac{P_{\min}}{\vartheta}$ time at the end of the round remains unused.

logical times. Intuitively, one can interpret the clock synchronization algorithm as generating ticks at "infinite" frequency , and we divide this frequency such that we obtain the desired period bounds for the pulse synchronization algorithm. We can divide by any value larger than $\mathcal{G}$; otherwise, pulses might overlap, which we ruled out for pulse synchronization.

Algorithm 9 Simulation of lock-step execution at $v \in V_g$, where δ_{in} and δ_{out} are its in- and outdegree, respectively, $\hat{\mathcal{S}}$ is an upper bound on $\mathcal{S}$, state and msg are the functions computing state updates and messages to send, and d_{msg} denotes the maximum message delay (excluding computations). A pulse synchronization algorithm is executed in parallel, which generates its first pulse after the simulation is initialized.

1: $s \leftarrow \langle$initial state$\rangle$ and $m_{\text{in}}[i] \leftarrow \perp$ for $i \in \{1, \dots, \delta_{\text{in}}\}$ ▷ initialization
2: $r \leftarrow 0$ ▷ round counter
3: $\tau \leftarrow \frac{P_{\min}}{1+\rho} - \hat{\mathcal{S}} - d_{\max,\text{send}}$ ▷ shorthand for constant
4: **while true do**
5: wait for next pulse at v
6: $h \leftarrow \text{getH}()$
7: $r \leftarrow r + 1$
8: **for** $i = 0, \dots, \delta_{\text{in}}(v) + \delta_{\text{out}}(v)$ in parallel **do**
9: **if** $i = 0$ **then** ▷ update state
10: $s \leftarrow \text{state}(s, m_{\text{in}})$
11: **else if** $1 \le i \le \delta_{in}(v)$ **then** ▷ receive messages
12: start receiving on incoming port i at local time $h + \frac{\tau - \hat{\mathcal{S}}}{1+\rho} + d_{\min,\text{send}}$
13: finish receiving on incoming port i at local time $h + P_{\min}$
14: let m be received message (use default if none or invalid)
15: $m_{\text{in}}[i] \leftarrow m$
16: **else** ▷ send messages
17: $j \leftarrow i - \delta_{in}(v)$
18: $m \leftarrow \text{msg}(s, m_{\text{in}}, j)$
19: wait until local time $h + \tau$
20: send m over outgoing port j
21: **end if**
22: **end for**
23: **end while**

Theorem 9.8. *Assume that for $T > \mathcal{G}$, the nodes in V_g run a clock synchronization algorithm with global skew $\mathcal{G}$, minimum clock rate* 1*, and maximum clock rate β. By adding only local computations to the algorithm, we can solve pulse synchronization with $\mathcal{S} = \mathcal{G}$, $P_{\min} = \frac{T-\mathcal{G}}{\beta}$, and $P_{\max} = T + \mathcal{G}$.*

Proof. The algorithm is simple. Each node $v \in V_g$ executes the clock synchronization algorithm, where w.l.o.g. we assume that logical clocks satisfy $L_v(0) \in [0, \mathcal{G}]$. In addition, a parallel thread executes the following:

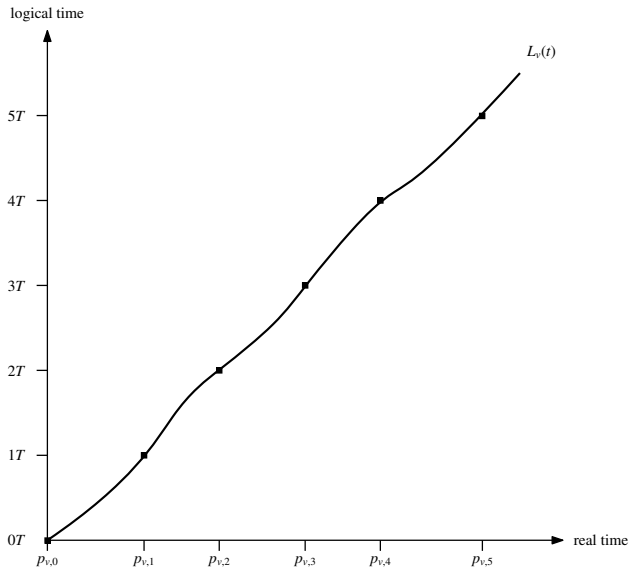

Figure 9.2
Relation between pulse times and logical clock at node $v \in V_g$ for Algorithm 10. Note that the logical clock rate varies between 1 and β, where typically $\beta - 1 \ll 1$. Hence the real time between pulses fluctuates slightly.

Algorithm 10 Pulse synchronization algorithm at $v \in V_g$ based on a clock synchronization algorithm. W.l.o.g., the code assumes that logical clocks satisfy $L_v(0) \in [0, \mathcal{G}]$.

1: $i \leftarrow 0$
2: **while true do**
3: wait until getL() = iT
4: generate i-th pulse
5: $i \leftarrow i + 1$
6: **end while**

We claim that Algorithm 10 satisfies the claims of the theorem. Because $T > \mathcal{G}$ and logical clocks are continuous and strictly increasing, v generates its i-th pulse at the unique time $p_{v,i}$ satisfying $L_v(p_{v,i}) = iT$. Fix any $i \in \mathbb{N}$ and $v, w \in V_g$. We check the three required properties.

1. W.l.o.g. over the choice of v and w, suppose $p_{v,i} \leq p_{w,i}$. We have that

Definition 9.4
Algorithm 10

$$\begin{aligned} L_w(p_{v,i} + \mathcal{G}) &\geq L_w(p_{v,i}) + \mathcal{G} \geq L_v(p_{v,i}) \\ &= iT = L_w(p_{w,i}), \end{aligned}$$

   implying that $p_{w,i} \leq p_{v,i} + \mathcal{G}$. Hence, for all $i \in \mathbb{N}$ and $v, w \in V_g$ it holds that $|p_{v,i} - p_{w,i}| \leq \mathcal{G}$, as claimed.
2. We have that

Definition 9.4
Algorithm 10

$$\begin{aligned} L_v\left(p_{w,i} + \frac{T - \mathcal{G}}{\beta}\right) &\leq L_v(p_{w,i}) - \mathcal{G} + T \leq L_w(p_{w,i}) + T \\ &= (i+1)T = L_v(p_{v,i+1}), \end{aligned}$$

   implying that $p_{v,i+1} \geq p_{w,i} + \frac{T-\mathcal{G}}{\beta}$. Hence, for each $i \in \mathbb{N}$ it holds that $\min_{v \in V_g}\{p_{v,i+1}\} - \max_{v \in V_g}\{p_{v,i}\} \geq \frac{T-\mathcal{G}}{\beta}$, as claimed.
3. We have that

Definition 9.4
Algorithm 10

$$\begin{aligned} L_v(p_{w,i} + \mathcal{G} + T) &\geq L_v(p_{w,i}) + \mathcal{G} + T \geq L_w(p_{w,i}) + T \\ &= (i+1)T = L_v(p_{v,i+1}), \end{aligned}$$

   implying that $p_{v,i+1} \leq p_{w,i} + \mathcal{G} + T$. Hence, for each $i \in \mathbb{N}$ it holds that $\max_{v \in V_g}\{p_{v,i+1}\} - \min_{v \in V_g}\{p_{v,i}\} \leq \mathcal{G} + T$, as claimed. □

We remark that Theorem 9.8 is "lossy" in terms of clock rates and clock skew. This is the result of using the pulse synchronization algorithm as a black box, using worst-case bounds. However, $P_{\max} - P_{\min}$ typically is (up to constants) bounded by the skew of the algorithm plus $(\vartheta - 1)P_{\max}$, meaning that we lost constant factors only.

9.3.4 From Pulse Synchronization to Clock Synchronization

Constructing a clock synchronization algorithm based on a pulse synchronization algorithm is conceptually not much harder, but technically more involved. Our goal is to linearly interpolate between the discrete clock steps provided by the pulse synchronization algorithm, where each pulse corresponds to between $P_{\min}$ and $P_{\max}$ time. We can not do this precisely, but will do so to the best degree possible based on the guarantees provided by the pulse synchronization algorithm and our local clocks.

Intuitively, we can view the pulse synchronization algorithm as providing a fault-tolerant distributed clock reference to which we digitally lock the logical clocks we generate, cf. **??**. In this view, $\frac{1}{P_{\min}}$ and $\frac{1}{P_{\max}}$ bound the frequency of the master clock given by the (local) pulses, and $\frac{dt}{dH_v}$ is the instantaneous frequency of the slave clock at $v \in V_g$, if it was free-running (i.e., not locked to

the reference). In line with this intuition, we need to specify a phase detector and a loop filter to complete the local PLL at node v. As we make sure that the logical clock as v is never running "too fast," the phase detector is given by checking how much behind the "target" value the logical clock is when a local pulse is generated. For the loop filter, we use a simple low pass filter, which spreads out catching up by the measured phase difference over $P_{\min}$ time. This simple scheme is sufficient to keep the output frequency between 1 and β for the smallest β we can manage.

Algorithm 11 Pulse synchronization algorithm at $v \in V_g$ based on a clock synchronization algorithm.

1: wait until initialization pulse
2: $\ell \leftarrow 0$ ▷ initialize logical clock
3: $i \leftarrow 0$ ▷ pulse number
4: $\ell^{\uparrow} \leftarrow 0$ ▷ clock increase to be amortized during current pulse
5: $h \leftarrow \text{getH}()$
6: **while true do**
7: wait until next pulse
8: $\ell \leftarrow i(1+\rho)P_{\max} + \text{getH}() - h$ ▷ logical clock value at pulse
9: $i \leftarrow i + 1$
10: $\ell^{\uparrow} \leftarrow i(1+\rho)P_{\max} - \ell$ ▷ difference to target value
11: $h \leftarrow \text{getH}()$
12: **end while**
13: **procedure** getL() ▷ returns $L_v(t)$ when called at time $t \geq p_{v,0}$
14: **return** $\ell + \text{getH}() - h + \ell^{\uparrow} \cdot \min\left\{\frac{\text{getH}()-h}{P_{\min}}, 1\right\}$
15: **end procedure**

We first show that Algorithm 11 computes logical clocks with rates between 1 and $\frac{\vartheta^2 P_{\max}}{P_{\min}}$.

Lemma 9.9 (Algorithm 11 computes logical clocks)**.** *For each $v \in V_g$, denote by $L_v\colon [p_{v,0}, \infty)$ the logical clock provided by the procedure* getL() *in Algorithm 11. Then for all times $t \geq t' \geq p_{v,0}$, it holds that $1 \leq L_v(t) - L_v(t') \leq \beta := \frac{\vartheta^2 P_{\max}}{P_{\min}}$.*

Proof. For each $i \in \mathbb{N}$, the function $\ell_v(p_{v,i}) + H_v(t) - h_v(p_{v,i}) + \ell_v^{\uparrow}(p_{v,i}) \cdot \min\left\{\frac{H_v(t)-h_v(p_{v,i})}{P_{\min}}, 1\right\}$ is continuous. It is also differentiable except at time

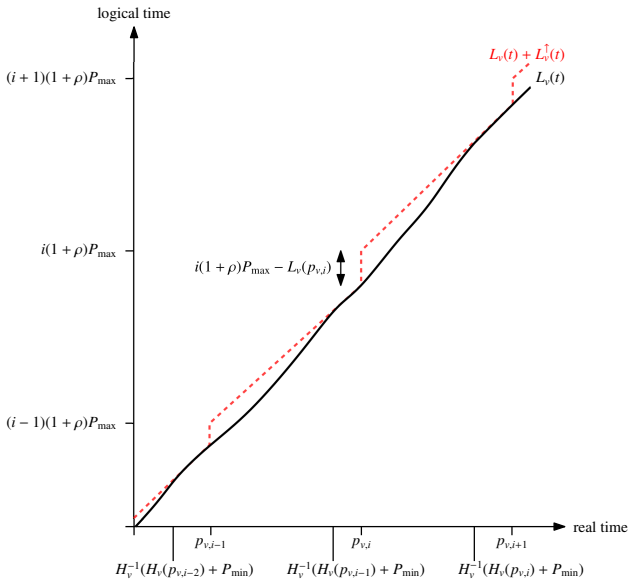

Figure 9.3
Logical clock at node $v \in V_g$ produced by Algorithm 11 as function of real time. The dashed red line indicates the "target value" of the logical clock, which advances by $\vartheta P_{\max}$ per pulse. The difference is amortized over $P_{\min}$ local time, which for the i-th pulse corresponds to $H_v^{-1}(H(p_{v,i}) + P_{\min}) - p_{v,i}$ real time.

$H_v^{-1}(H(p_{v,i}) + P_{\min})$. Observe that for all $i \in \mathbb{N}$, we have that

$\frac{dH_v}{dt} \geq 1$, Definition 9.5

$$\begin{aligned} H_v(p_{v,i+1}) - H_v(p_{v,i}) &\geq p_{v,i+1} - p_{v,i} \\ &\geq P_{\min}. \end{aligned} \tag{9.5}$$

Hence, it holds for each $i \in \mathbb{N}$ that evaluating the logical clock of v at time $p_{v,i+1}$ results in

$$\begin{aligned} & L_v(p_{v,i+1}) \\ \text{Lines 11 and 14} \quad &= \ell_v(p_{v,i+1}) \\ \text{Lines 7 to 11} \quad &= \ell_v(p_{v,i}) + H_v(p_{v,i+1}) - h_v(p_{v,i}) + \ell^{\uparrow}(p_{v,i}) \\ (9.5) \quad &= \ell_v(p_{v,i}) + H_v(p_{v,i+1}) - h_v(p_{v,i}) + \ell^{\uparrow}(p_{v,i}) \cdot \min\left\{ \frac{H_v(p_{v,i+1}) - h_v(p_{v,i})}{P_{\min}}, 1 \right\}. \end{aligned}$$

We conclude that L_v is continuous. Moreover, it is differentiable at all times except possibly $p_{v,i}$ and $H_v^{-1}(H_v(p_{v,i}) + P_{\min})$ for each $i \in \mathbb{N}$. Therefore, it suffices to show that the derivative takes on values between 1 and β at times when it exists.

Accordingly, consider any time t when the derivative is guaranteed to exist. There are two cases. If $H_v(t) - h_v(t) > P_{\min}$, then simply

$$\frac{dL_v}{dt}(t) = \frac{dH_v}{dt}(t) \in [1, \vartheta]. \tag{9.6}$$

On the other hand, if $H_v(t) - h_v(t) < P_{\min}$, let $i \in \mathbb{N}$ be maximal such that $p_{v,i} < t$. Then

$$\frac{dL_v}{dt}(t) = \frac{dH_v}{dt}(t) + \frac{\ell_v^{\uparrow}(p_{v,i})}{P_{\min}} \cdot \frac{dH_v}{dt}(t) = \frac{dH_v}{dt}(t) \cdot \frac{\ell_v^{\uparrow}(p_{v,i}) + P_{\min}}{P_{\min}}. \tag{9.7}$$

We claim that

$$0 \leq \ell_v^{\uparrow}(p_{v,i}) \leq \vartheta P_{\max} - P_{\min} \tag{9.8}$$

for all $i \in \mathbb{N}$. This is trivially satisfied for $i = 0$, so consider $i \neq 0$. We have that

$$\begin{aligned} \ell_v^{\uparrow}(p_{v,i}) &= i_v(p_{v,i})\vartheta P_{\max} - \ell_v(p_{v,i}) && \text{Line 10} \\ &= \vartheta P_{\max} - (H_v(p_{v,i}) - H_v(p_{v,i-1})). && \text{Lines 7 to 9 and 11} \end{aligned}$$

The claim now follows by observing that

$$\begin{aligned} P_{\min} &\leq p_{v,i} - p_{v,i-1} && \text{Definition 9.5} \\ &\leq H_v(p_{v,i}) - H_v(p_{v,i-1}) && \tfrac{dH_v}{dt} \geq 1 \\ &\leq \vartheta(p_{v,i} - p_{v,i-1}) && \tfrac{dH_v}{dt} \leq \vartheta \\ &\leq \vartheta P_{\max}. && \text{Definition 9.5} \end{aligned}$$

We conclude that

$$\begin{aligned} 1 \leq \frac{dH_v}{dt}(t) &= \min\left\{\frac{dH_v}{dt}(t), \frac{dH_v}{dt}(t) \cdot \frac{\ell_v^{\uparrow}(p_{v,i}) + P_{\min}}{P_{\min}}\right\} && (9.8) \\ &\leq \frac{dL_v}{dt}(t) \leq \max\left\{\frac{dH_v}{dt}(t), \frac{dH_v}{dt}(t) \cdot \frac{\sup_{i\in\mathbb{N}}\{\ell_v^{\uparrow}(p_{v,i})\} + P_{\min}}{P_{\min}}\right\} && (9.6), (9.7) \\ &\leq \max\left\{\frac{dH_v}{dt}(t), \frac{dH_v}{dt}(t) \cdot \frac{\vartheta P_{\max}}{P_{\min}}\right\} && (9.8) \\ &= \frac{dH_v}{dt}(t) \cdot \frac{\vartheta P_{\max}}{P_{\min}} \leq \frac{\vartheta^2 P_{\max}}{P_{\min}} = \beta. && \square P_{\max} \geq P_{\min} \end{aligned}$$

Next, we bound the skew of the clocks generated by Algorithm 11.

Lemma 9.10 (Skew of Algorithm 11)**.** *Set* $\beta := \frac{\vartheta^2 P_{\max}}{P_{\min}}$ *as in Lemma 9.9 and assume that* $\max_{v\in V_g}\{p_{v,0}\} \leq 0$*. Then the clocks* L_v *provided by Algorithm 11*

guarantee

$$\mathcal{G} \leq (\vartheta - 1)P_{\max} + \beta\mathcal{S}.$$

Proof. Fix $v, w \in V_g$. Consider a time $t \geq \max\{p_{v,0}, p_{w,0}\}$. From Definition 9.5, it is immediate that $p_{w,i_v(t)+1} > p_{v,i_v(t)}$ and $p_{v,i_w(t)+1} > p_{w,i_w(t)}$, implying that $i := \max\{i_v(t), i_w(t)\} \geq \min\{i_v(t), i_w(t)\} - 1$.

Consider first the case that $i_w(t) = i-1$, i.e., $t \in [p_{v,i}, p_{w,i}) \subseteq [p_{v,i}, p_{v,i}+\mathcal{S})$. Note that

$\frac{dH_w}{dt} \geq 1$, Definition 9.5

$$H_w(t) - H_w(p_{w,i-1}) \geq t - p_{w,i-1} \geq p_{v,i} - p_{w,i-1} \geq P_{\min}. \tag{9.9}$$

Hence, we can bound

$$\begin{aligned}
&L_v(t) - L_w(t)\\
&= H_v(t) - H_v(p_{v,i-1}) + \ell_v^{\uparrow}(p_{v,i}) \cdot \max\left\{\frac{H_v(t) - H_v(p_{v,i})}{P_{\min}}, 1\right\}\\
&\qquad - (H_w(t) - H_w(p_{w,i-1}))\\
&\leq \vartheta(t - p_{v,i-1}) + \ell_v^{\uparrow}(p_{v,i}) \cdot \frac{\vartheta(t - p_{v,i})}{P_{\min}} - (t - p_{w,i-1})\\
&= (\vartheta - 1)(t - p_{v,i-1}) + (\vartheta P_{\max} - (H_v(p_{v,i}) - H_v(p_{v,i-1}))) \cdot \frac{\vartheta\mathcal{S}}{P_{\min}} + \mathcal{S}\\
&= (\vartheta - 1)(t - p_{v,i-1}) + (\vartheta P_{\max} - (p_{v,i} - p_{v,i-1})) \cdot \frac{\vartheta\mathcal{S}}{P_{\min}} + \mathcal{S}\\
&< (\vartheta - 1)P_{\max} + \left(1 + \frac{(\vartheta^2 P_{\max} - \vartheta P_{\min})}{P_{\min}}\right)\mathcal{S} < (\vartheta - 1)P_{\max} + \beta\mathcal{S}.
\end{aligned} \tag{9.10, 9.11}$$

Line 14, (9.9)

$1 \leq \frac{dH}{dt} \leq \vartheta$, (9.8)

Lines 7 to 11

$\frac{dH_v}{dt} \geq 1$

Definition 9.5

By continuity of the logical clock functions, this bound also applies to $t = p_{w,i} = \max\{p_{v,i}, p_{w,i}\}$, which will be useful later.

The second case is that $i_v(t) = i - 1$, i.e., $t \in [p_{w,i}, p_{v,i}) \subseteq [p_{w,i}, p_{w,i} + \mathcal{S})$. Analogously to the first case, we can show Equation (9.9) for v, yielding

(9.8), (9.10)

$$\begin{aligned}
L_v(t) - L_w(t) &\leq H_v(t) - H_w(p_{v,i-1}) - (H_w(t) - H_w(p_{v,i-1}))\\
&\leq \vartheta(t - p_{v,i-1}) - (t - p_{w,i-1})\\
&= (\vartheta - 1)(t - p_{v,i-1}) + \vartheta(p_{w,i-1} - p_{v,i-1})\\
&\leq (\vartheta - 1)P_{\max} + \vartheta\mathcal{S} < (\vartheta - 1)P_{\max} + \beta\mathcal{S}.
\end{aligned}$$

$1 \leq h_v, h_w \leq \vartheta$

Definition 9.5

Again, this bound also applies to $t = p_{v,i} = \max\{p_{v,i}, p_{w,i}\}$.

The final case is that $i = i_v(t) = i_w(t)$, from which we can conclude that $t \in [\max\{p_{v,i}, p_{w,i}\}, \min\{p_{v,i+1}, p_{w,i+1}\}]$. We get that

$$
\begin{aligned}
& L_v(t) - L_w(t) \\
&= L_v(t) - L_v(p_{v,i}) + L_v(p_{v,i}) - L_w(p_{w,i}) - (L_w(t) - L_w(p_{w,i})) && \text{adding 0} \\
&= L_v(t) - L_v(p_{v,i}) - \ell_v^{\uparrow}(p_{v,i}) + \ell_w^{\uparrow}(p_{w,i}) - (L_w(t) - L_w(p_{w,i})) && \text{Lines 8 and 10} \\
&= H_v(t) - H_v(p_{v,i}) - \ell_v^{\uparrow}(p_{v,i}) \cdot \max\left\{1 - \frac{H_v(t) - H_v(p_{v,i})}{P_{\min}}, 0\right\} \\
&\qquad - \left(H_w(t) - H_w(p_{w,i}) - \ell_w^{\uparrow}(p_{w,i}) \cdot \max\left\{1 - \frac{H_w(t) - H_w(p_{w,i})}{P_{\min}}, 0\right\}\right) && \text{Line 14} \\
&\le \vartheta(t - p_{v,i}) - \ell_v^{\uparrow}(p_{v,i}) \cdot \max\left\{1 - \frac{\vartheta(t - p_{v,i})}{P_{\min}}, 0\right\} \\
&\qquad - \left((t - p_{w,i}) - \ell_w^{\uparrow}(p_{w,i}) \cdot \max\left\{1 - \frac{t - p_{w,i}}{P_{\min}}, 0\right\}\right) && (9.12) \quad 1 \le \tfrac{dH}{dt} \le \vartheta
\end{aligned}
$$

This last expression is piece-wise linear as function of t, implying that the maximum is attained at some

$$t \in \left\{\max\{p_{v,i}, p_{w,i}\}, p_{v,i} + \frac{P_{\min}}{\vartheta}, p_{w,i} + P_{\min}, \min\{p_{v,i+1}, p_{w,i+1}\}\right\}.$$

Observe that the previous cases already cover the option $t = \max\{p_{v,i}, p_{w,i}\}$, and by applying them for index $i+1$ the same holds for $t = \min\{p_{v,i+1}, p_{w,i+1}\}$.

Next, we can bound

$$
\begin{aligned}
& L_v\left(p_{v,i} + \frac{P_{\min}}{\vartheta}\right) - L_w\left(p_{v,i} + \frac{P_{\min}}{\vartheta}\right) \\
&\le \frac{(\vartheta - 1)P_{\min}}{\vartheta} + p_{w,i} - p_{v,i} + \ell_w^{\uparrow}(p_{w,i}) \cdot \max\left\{1 - \frac{\vartheta(p_{v,i} - p_{w,i}) + P_{\min}}{\vartheta P_{\min}}, 0\right\} && (9.12) \\
&= \frac{(\vartheta - 1)P_{\min}}{\vartheta} + p_{w,i} - p_{v,i} + (\vartheta P_{\max} - (H_w(p_{w,i}) - H_w(p_{w,i-1}))) \\
&\qquad \cdot \max\left\{\frac{(\vartheta - 1)P_{\min} + \vartheta(p_{w,i} - p_{v,i})}{\vartheta P_{\min}}, 0\right\} && \text{Lines 7 to 11} \\
&= \frac{(\vartheta - 1)P_{\min}}{\vartheta} + p_{w,i} - p_{v,i} + (\vartheta P_{\max} - P_{\min}) \\
&\qquad \cdot \max\left\{\frac{(\vartheta - 1)P_{\min} + \vartheta(p_{w,i} - p_{v,i})}{\vartheta P_{\min}}, 0\right\} && \tfrac{dH_w}{dt} \ge 1, \text{ Definition 9.5} \\
&\le \frac{(\vartheta - 1)P_{\min}}{\vartheta} + \mathcal{S} + (\vartheta P_{\max} - P_{\min}) \cdot \frac{(\vartheta - 1)P_{\min} + \vartheta\mathcal{S}}{\vartheta P_{\min}} && \text{Definition 9.5} \\
&= (\vartheta - 1)P_{\max} + \left(1 + \frac{\vartheta P_{\max} - P_{\min}}{P_{\min}}\right)\mathcal{S} < (\vartheta - 1)P_{\max} + \beta\mathcal{S}.
\end{aligned}
$$

Last, we compute

(9.8), (9.12) Definition 9.5
$$\begin{aligned} L_v(p_{w,i} + P_{\min}) - L_w(p_{w,i} + P_{\min}) &\leq (\vartheta - 1)P_{\min} + p_{w,i} - p_{v,i} \\ &\leq (\vartheta - 1)P_{\min} + \mathcal{S} < (\vartheta - 1)P_{\max} + \beta\mathcal{S}. \end{aligned}$$

As we bounded $L_v(t) - L_w(t)$ from above in all cases and $v, w \in V_g$ and t were arbitrary, we can conclude that indeed $\mathcal{G} \leq (\vartheta - 1)P_{\max} + \beta\mathcal{S}$. □

Theorem 9.11. *Suppose the nodes in V_g run a pulse synchronization algorithm with skew $\mathcal{S}$, minimum period $P_{\min}$, and maximum period $P_{\max}$. Furthermore, assume that node $v \in V_g$ generates its first pulse at real time $p_{v,0} \in [-\mathcal{S}, 0]$. By adding only local computations to the algorithm, we can solve clock synchronization with global skew $(\vartheta - 1)P_{\max} + \beta\mathcal{S}$, minimum clock rate 1, and maximum clock rate $\beta := \frac{\vartheta^2 P_{\max}}{P_{\min}}$.*

Proof. We claim that Algorithm 11 satisfies the claims of the theorem. By Lemma 9.9, the algorithm provides logical clocks satisfying the rate bounds, and by Lemma 9.10, the skew bound holds. □

Note that, by choosing T large enough, Theorem 9.11 preserves frequency arbitrarily well. On the other hand, we can choose T in $\Theta(\mathcal{G})$, maintaining asymptotically optimal speed of simulation of lock-step execution.

9.4 Impossibility of Synchronization with one Third of Faulty Nodes

Before delving into the impossibility proof, the reader is encouraged to spend a few minutes to form an idea on how faulty nodes may disrupt algorithms.

E9.11 Argue that synchronization cannot be guaranteed if half or more of the nodes are Byzantine faulty.

E9.12 Argue that a node for which half or more of its neighbors are faulty cannot reliably synchronize to correct nodes that are not its neighbors.

So, where does the smaller threshold of $3f < n$ come from? Intuitively, the issue is that Byzantine nodes can play different sets of nodes differently. Instead of simply drowning out the information from correct nodes by being in the majority, they can follow a divide-and-conquer approach. As always, we will make an indistinguishability argument, but this time there will always be some correct nodes who can observe a difference. The issue is that they cannot *prove* to the other correct nodes that it is not them who are faulty.

We prove the lower bound for the pulse synchronization task. We partition the node set into three sets $A, B, C \subset V$ so that $1 \leq |A|, |B|, |C| \leq f$. We will

	$H_A(t)$	$H_B(t)$	$H_C(t)$
$\mathcal{E}_0$	νt	$\nu^2 t$	$\leftarrow$ arbitrary $t \rightarrow$
$\mathcal{E}_1$	$\nu^2 t$	$\leftarrow \nu^3 t$ $t \rightarrow$	νt
$\mathcal{E}_2$	$\leftarrow \nu^3 t$ $t \rightarrow$	νt	$\nu^2 t$
$\mathcal{E}_3$	νt	$\nu^2 t$	$\leftarrow \nu^3 t$ $t \rightarrow$
$\mathcal{E}_4$	$\nu^2 t$	$\leftarrow \nu^3 t$ $t \rightarrow$	νt
$\mathcal{E}_5$	$\leftarrow \nu^3 t$ $t \rightarrow$	νt	$\nu^2 t$
$\mathcal{E}_6$	νt	$\nu^2 t$	$\leftarrow \nu^3 t$ $t \rightarrow$
...	...	...	...

Table 9.1
Hardware clock speeds in the different executions for the different sets. The red entries indicate faulty sets, simulating a clock speed of $\nu^3 t$ to the set "to the left" and t to the set "to the right." For $k \in \mathbb{N}_0$, execution pairs $(\mathcal{E}_{3k}, \mathcal{E}_{3k+1})$ are indistinguishable to nodes in A, pairs $(\mathcal{E}_{3k+1}, \mathcal{E}_{3k+2})$ are indistinguishable to nodes in C, and pairs $(\mathcal{E}_{3k+2}, \mathcal{E}_{3k+3})$ are indistinguishable to nodes in B. That is, in $\mathcal{E}_i$ faulty nodes mimic the behavior they have in $\mathcal{E}_{i-1}$ to the set left of them, and that from $\mathcal{E}_{i+1}$ to the set to the right.

construct a sequence of executions showing that either synchronization is lost in some execution (i.e., any finite skew bound $\mathcal{S}$ is violated) or the algorithm cannot guarantee bounds on the period, cf. Table 9.1. In each execution, one of the sets, say A, consists entirely of faulty nodes. All of the nodes in the (correct) set B will have identical hardware clocks, as will the nodes in C. The faulty nodes in A attempt to fool the correct nodes in B and C as follows: to one set, say B, faulty nodes send messages to each $v \in B$ that lead v to believe that v's clock is fast. Similarly, nodes in A try to convince each $w \in C$ that w's clock is slow. All clock rates (actual or simulated) will lie between 1 and ν^3, where $\nu > 1$ is small enough so that $\nu^3 \leq \vartheta$ and $d \leq \nu^3(d-u)$. This way, message delays can be chosen such that messages arrive at the same local times without violating message delay bounds.

For each pair of consecutive executions, the executions are indistinguishable to the node set that is correct in both executions *and* there is a factor of $\nu > 1$ between the speeds of hardware clocks. This means that the pulses are generated at by factor ν higher speed. However, as the skew bounds are to be satisfied, the set of correct nodes that *know* that something is different will have to generate

pulses faster. Thus, in execution $\mathcal{E}_i$, pulses are generated at an amortized rate of (at least) $\nu^i P_{\min}$. For $i > \log_\nu \frac{P_{\max}}{P_{\min}}$, we arrive at a contradiction.

Lemma 9.12. *Suppose $3 \leq n \leq 3f$. Then, for any algorithm $\mathcal{A}$, there exists $\nu > 1$ and a sequence of executions $\mathcal{E}_i$, $i \in \mathbb{N}_0$, with the properties stated in Table 9.1.*

Proof. Choose $\nu := \min\left\{\vartheta, \frac{d}{d-u}\right\}^{1/3}$. We construct the entire sequence concurrently, where we advance real time in execution $\mathcal{E}_i$ at speed ν^{-i}. All correct nodes run $\mathcal{A}$, which specifies the local times at which these nodes send messages as well as their content. We maintain the invariant that the constructed parts of the executions satisfy the stated properties. In particular, this defines the hardware clocks of correct nodes at all times. Any message a node v (faulty or not) sends at time t to some node w is received at local time $H_w(t) + d$. By the choice of ν, this means that all hardware clock rates (of correct nodes) and message delays are within the required bounds, i.e., all constructed executions are feasible.

We need to specify the messages sent by faulty nodes in a way that achieves the desired indistinguishability. To this end, consider the set of faulty nodes in execution $\mathcal{E}_i$, $i \in \mathbb{N}_0$. If in execution $\mathcal{E}_{i+1}$ such a node v sends a message to some w in the "right" set (i.e., B is right of A, C of B, and A of C) at time $t = \frac{H_v(t)}{\nu}$, it sends the same message in $\mathcal{E}_i$ at time νt. Thus, it is received at local time

$$H_w^{(\mathcal{E}_i)}(\nu t) + d = \nu^2 t + d = H_w^{\mathcal{E}_{i+1}}(t) + d.$$

Similarly, consider the set of faulty nodes in execution $\mathcal{E}_i$, $i \in \mathbb{N}$. If in execution $\mathcal{E}_{i-1}$ a node v from this set sends a message to some w in the "left" set (i.e., A is left of B, B of C, and C or A) at time t, it sends the same message in $\mathcal{E}_i$ at time $\frac{t}{\nu}$. Thus, it is received at local time

$$H_w^{(\mathcal{E}_i)}\left(\frac{t}{\nu}\right) + d = \nu t + d = H_w^{(\mathcal{E}_{i-1})}(t) + d.$$

Together, this implies that for $k \in \mathbb{N}_0$, execution pairs $(\mathcal{E}_{3k}, \mathcal{E}_{3k+1})$ are indistinguishable to nodes in A, pairs $(\mathcal{E}_{3k+1}, \mathcal{E}_{3k+2})$ are indistinguishable to nodes in C, and pairs $(\mathcal{E}_{3k+2}, \mathcal{E}_{3k+3})$ are indistinguishable to nodes in B, as claimed. Note that it does not matter which messages are sent from the nodes in C to nodes in B in execution $\mathcal{E}_0$; for example, we can rule that they send no messages to nodes in B at all.

It might seem as if the proof were complete. However, each execution is defined in terms of others, so it is not entirely clear that the above assignment is possible. This is where we use the aforementioned approach of "constructing execution $\mathcal{E}_i$ at speed ν^{-i}." Think of each faulty node as simulating two virtual

nodes, one for messages sent "to the left," which has local time $\nu^3 t$ at time t, and one for messages sent "to the right," which has local time t at time t. This way, there is a one-to-one correspondence between the virtual nodes of a faulty node v in execution $\mathcal{E}_i$ and the corresponding nodes in executions $\mathcal{E}_{i-1}$ and $\mathcal{E}_{i+1}$, respectively (up to the case $i = 0$, where the "left" virtual nodes do not send messages). If a faulty node v needs to send a message in execution $\mathcal{E}_i$, the respective virtual node sends the message at the same local time as v sends the message in execution $\mathcal{E}_{i-1}$ (left) or $\mathcal{E}_{i+1}$ (right). In terms of real time, there is exactly a factor of ν: if v is faulty in $\mathcal{E}_i$ and wants to determine the behavior of its virtual node corresponding to $\mathcal{E}_{i-1}$ up to time t, it needs to simulate $\mathcal{E}_{i-1}$ up to time νt; similarly, when doing the same for its virtual node corresponding to $\mathcal{E}_{i+1}$, it needs to simulate $\mathcal{E}_{i+1}$ up to time $\frac{t}{\nu}$. Thus, when simulating all executions concurrently, where $\mathcal{E}_i$ progresses at rate ν^{-i}, at all times the behavior of faulty nodes according to the above scheme can be determined. This completes the proof. □

Theorem 9.13. *Pulse synchronization is impossible if* $3 \leq n \leq 3f$.

Proof. Assume for contradiction that there is an algorithm solving pulse synchronization. We apply Lemma 9.12, yielding a sequence of executions $\mathcal{E}_i$ with the properties stated in Table 9.1. We will show that pulses are generated arbitrarily fast, contradicting the minimum period requirement. We show this by induction on $i \in \mathbb{N}_0$, where the induction hypothesis is that there is some $v \in V_g^{(\mathcal{E}_i)}$ satisfying that

$$p_{v,j}^{(\mathcal{E}_i)} - p_{v,1}^{(\mathcal{E}_i)} \leq (j-1)\nu^{-i} P_{\max} + 2i\mathcal{S}$$

for all $j \in \mathbb{N}$, where $\nu > 1$ is given by Lemma 9.12. This is trivial for the base case $i = 0$ by the maximum period requirement.

For the induction step from i to $i+1$, let $v \in V_g^{(\mathcal{E}_i)}$ be a node with $p_{v,j}^{(\mathcal{E}_i)} - p_{v,1}^{(\mathcal{E}_i)} \leq (j-1)\nu^{-i} P_{\max} + 2i\mathcal{S}$ for all $j \in \mathbb{N}_0$. Let $w \in V_g^{(\mathcal{E}_i)} \cap V_g^{(\mathcal{E}_{i+1})}$ be a node that is correct in both $\mathcal{E}_i$ and $\mathcal{E}_{i+1}$. By the skew bound,

$$p_{w,j}^{(\mathcal{E}_i)} - p_{w,1}^{(\mathcal{E}_i)} \leq p_{v,j}^{(\mathcal{E}_i)} - p_{v,1}^{(\mathcal{E}_i)} + 2\mathcal{S} \leq (j-1)\nu^{-i} P_{\max} + 2(i+1)\mathcal{S}$$

for all $j \in \mathbb{N}$. By Lemma 9.12, w cannot distinguish between $\mathcal{E}_i$ and $\mathcal{E}_{i+1}$. Because $H_w^{(\mathcal{E}_{i+1})}(t/\nu) = \nu t = H_w^{(\mathcal{E}_{i+1})}(t)$, we conclude that $p_{w,j}^{(\mathcal{E}_{i+1})} = \nu^{-1} p_{w,j}^{(\mathcal{E}_i)}$ for all $j \in \mathbb{N}$. Hence,

$$p_{w,j}^{(\mathcal{E}_{i+1})} - p_{w,1}^{(\mathcal{E}_{i+1})} \leq \nu^{-1}\left(p_{w,j}^{(\mathcal{E}_i)} - p_{w,1}^{(\mathcal{E}_i)}\right) \leq (j-1)\nu^{-(i+1)} P_{\max} + 2(i+1)\mathcal{S}$$

for all $j \in \mathbb{N}$, completing the induction step.

Now choose $i \in \mathbb{N}$ large enough so that $\nu^{-i} P_{\max} < P_{\min}$ and let $v \in V_g^{(\mathcal{E}_i)}$ be a node to which the claim applies in $\mathcal{E}_i$. Choosing $j - 1 > 2i\mathcal{S}(P_{\min} - \nu^{-i} P_{\max})$, it follows that

$$p_{v,j}^{(\mathcal{E}_i)} - p_{v,1}^{(\mathcal{E}_i)} \leq (j-1)\nu^{-i} P_{\max} + 2i\mathcal{S} < (j-1) P_{\min} .$$

Hence, the minimum period bound is violated, as there must be some index $j' \in \{1, \dots, j-1\}$ for which $p_{v,j'+1}^{(\mathcal{E}_i)} - p_{v,j'}^{(\mathcal{E}_i)} < P_{\min}$. □

E9.13 Solve pulse synchronization for $n = 2$ and $f = 1$.

The above theorem concerns pulse synchronization. However, we established that pulse and clock synchronization are very closely related, so the impossiblity of solving pulse synchronization implies the same for clock synchronization.

Corollary 9.14. *Clock synchronization is impossible if* $3 \leq n \leq 3f$ *and any (amortized) lower bound on clock rates is to be guaranteed.*

E9.14 Prove the corollary. Note that the corollary does *not* require an upper bound on clock rates. To reach a contradiction, show that no finite skew bound can be satisfied.

9.5 Pulse Synchronization with less than one Third of Faulty Nodes

We have seen that we cannot hope to find an algorithm that always works if $n \leq 3f$. We now show that $n > 3f$ is not only sufficient, but a *simple* algorithm can deal with this setting, despite the arbitrary behavior of faulty nodes. This simplicity enables us to describe the algorithm as a small state machine, plus one memory flag at each node for each incoming link. Concretely, we make the following restrictions (which do not constrain the behavior of faulty nodes!):

- Nodes communicate by broadcast (i.e., sending the same information to all nodes, including themselves). Note that faulty nodes do not need to obey this rule!
- Messages are trivial: Nodes communicate only when they transition to state PROPOSE.
- Each node v has a memory *flag* for every node $w \in V$ (including v itself), indicating whether v received such a message from w in the current iteration of the loop in the state machine. On some state transitions, v will reset all of its flags to 0, indicating that it starts a new iteration locally, in which it has not yet received any propose messages.
- Not accounting for the memory flags, each node runs a state machine with a constant number of states.

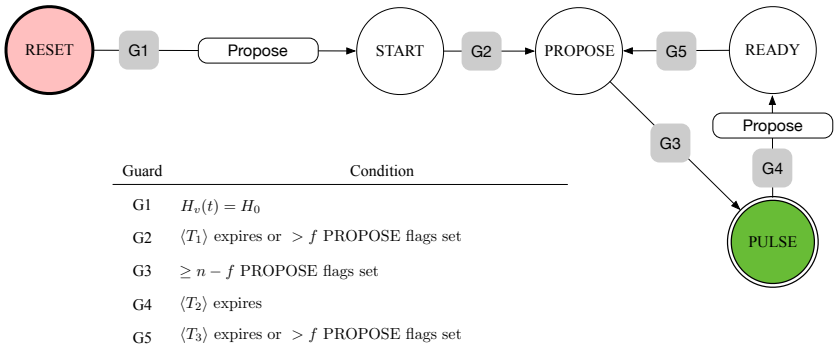

Figure 9.4
State machine of a node in the pulse synchronisation algorithm. State transitions occur when the condition of the guard in the respective edge is satisfied (gray boxes). All transition guards involve checking whether a local timer expires or a node has received PROPOSE messages from sufficiently many different nodes. The only communication is that a node broadcasts to all nodes (including itself) when it transitions to PROPOSE. The notation $\langle T \rangle$ evaluates to **true** when T time units have passed on the local clock since the transition to the current state. The boxes labeled PROPOSE indicate that a node clears its PROPOSE memory flags when transitioning from RESET to START or from PULSE to READY. That is, the node forgets who it has "seen" in PROPOSE at some point in the previous iteration. All nodes initialize their state machine to state RESET, which they leave at the time t when $H_v(t) = H_0$. Whenever a node transitions to state PULSE, it generates a pulse. The constraints imposed on the timeouts are listed in Inequalities (9.13)–(9.16).

- Transitions in this state machine are triggered by expressions involving (i) the own state, (ii) thresholds for the number of memory flags that are set (i.e., 1), and (iii) timeouts. A timeout means that a node waits for a certain amount of local time after entering a state before considering the timeout *expired*, i.e., evaluating the respective expression to **true**. The only exception is the starting state RESET, from which nodes transition to START when the local clock reaches H_0, where we assume that $\max_{v \in V_g} \{H_v(0)\} < H_0$. This corresponds to some (possibly weak) guarantee of synchronization at initialization.

The algorithm, from the perspective of a node, is depicted in Figure 9.4.

E9.15 Try to figure out the intended operation of the algorithm. Assume first that all correct nodes start in state READY.

The idea is to repeat the following cycle:

- At the beginning of an iteration, all nodes transition to state READY (or, initially, START) within a bounded time span. This resets the flags.

- Nodes wait in this state until they are sure that all correct nodes reached it. Then, when a local timeout expires, they transition to PROPOSE.
- When it looks like all correct nodes (may) have arrived there, they transition to PULSE. As the faulty nodes might refuse to send any messages, this means to wait for $n - f$ nodes having announced to be in PROPOSE.
- However, faulty nodes may also sent PROPOSE messages, meaning that the threshold is reached despite some nodes still waiting in READY for their timeouts to expire. To "pull" such stragglers along, nodes will also transition to PROPOSE if more than f of their memory flags are set. This is proof that at least one correct node transitioned to PROPOSE due to its timeout expiring, so no "early" transitions are caused by this rule.
- Thus, if *any* node hits the $n - f$ threshold, no more than d time later *each* node will hit the $f + 1$ threshold. Another d time later all nodes hit the $n - f$ threshold, i.e., the algorithm has skew $2d$.
- The nodes wait in PULSE sufficiently long to ensure that no PROPOSE messages are in transit any more before transitioning to READY and starting the next iteration.

For this reasoning to work out, a number of timing constraints need to be satisfied:

$$H_0 > \max_{v \in V_g} \{H_v(0)\} \tag{9.13}$$

$$\frac{T_1}{\vartheta} \geq H_0 \tag{9.14}$$

$$\frac{T_2}{\vartheta} \geq 3d \tag{9.15}$$

$$\frac{T_3}{\vartheta} \geq \left(1 - \frac{1}{\vartheta}\right) T_2 + 2d \tag{9.16}$$

We first show that the propose-pull mechanism works as intended, provided it is set up correctly, i.e., all correct nodes transition to START or READY within a small time window.

Lemma 9.15. *Suppose $3f < n$, $\Delta \geq 0$, and the above constraints are satisfied. Moreover, assume that each $v \in V_g$ transitions to START (READY) at a time $t_v \in [t - \Delta, t]$, no such node transitions to PROPOSE during $(t - \Delta - d, t_v)$, and $T_1 \geq \vartheta\Delta$ ($T_3 \geq \vartheta\Delta$). Then there is a time $t' \in \left(t - \Delta + \frac{T_1}{\vartheta}, t + T_1 - d\right)$ ($t' \in \left(t - \Delta + \frac{T_3}{\vartheta}, t + T_3 - d\right)$) such that each $v \in V_g$ transitions to PULSE during $[t', t' + 2d)$.*

Proof. We perform the proof for the case of START and T_1; the other case is analogous. Let t_p denote the smallest time larger than $t - \Delta - d$ when some $v \in V_g$ transitions to PROPOSE (such a time exists, as T_1 will expire if a node does not transition to PROPOSE before this happens). By assumption and the definition of t_p, no $v \in V_g$ transitions to PROPOSE during $(t-\Delta-d, t_p)$, implying that no node receives a message from any such node during $[t - \Delta, t_p]$. As $v \in V_g$ clears its memory flags when transitioning to READY at time $t_v \geq t - \Delta$, this implies that the node(s) from V_g that transition to PROPOSE at time t_p do so because T_1 expired. As hardware clocks run at most at rate ϑ and for each $v \in V_g$ it holds that $t_v \geq t - \Delta$, it follows that

$$t_p \geq t - \Delta + \frac{T_1}{\vartheta} \geq t\,.$$

Thus, at time $t_p \geq t$, each $v \in V_g$ has reached state READY and will not reset its memory flags again without transitioning to PULSE first. Therefore, each $v \in V_g$ will transition to PULSE: Each $v \in V_g$ transitions to PROPOSE during $[t_p, t + T_1]$, as it does so at the latest at time $t_v + T_1 \leq t + T_1$ due to T_1 expiring. Thus, by time $t + T_1 + d$ each $v \in V_g$ received the respective messages and, as $|V_g| \geq n - f$, transitioned to PULSE.

It remains to show that all correct nodes transition to PULSE within $2d$ time. Let t' be the minimum time after t_p when some $v \in V_g$ transitions to PULSE. If $t' \geq t + T_1 - d$, the claim is immediate from the above observations. Otherwise, note that out of the $n - f$ of v's flags that are **true**, at least $n - 2f > f$ correspond to nodes in V_g. The messages causing them to be set have been sent at or after time t_p, as we already established that any flags that were raised earlier have been cleared before time $t \leq t_p$. Their senders have broadcasted their transition to PROPOSE to all nodes, so any $w \in V_g$ has more than f flags raised by time $t' + d$, where d accounts for the potentially different travelling times of the respective messages. Hence, each $w \in V_g$ transitions to PROPOSE before time $t' + d$, the respective messages are received before time $t' + 2d$, and, as $|V_g| \geq n - f$, each $w \in V_g$ transitions to PULSE during $[t', t' + 2d)$. □

E9.16 Show a tight bound on the size of the time window during which the transitions to PULSE occur, taking into account that messages are under way for at least u time.

Lemma 9.16. *Suppose $3f < n$ and the constraints of Equations (9.13) to (9.16) are satisfied. Then the algorithm given in Figure 9.4 solves the pulse synchronization problem with $\mathcal{S} = 2d$, $P_{\min} = \frac{T_2+T_3}{\vartheta} - 2d$, and $P_{\max} = T_2 + T_3 + 3d$.*

Proof. We prove the claim by induction on the pulse number. For each pulse, we invoke Lemma 9.15. The first time, we use that all nodes start with hardware clock values in the range $[0, H_0)$ by (9.13). As hardware clocks run at least at rate 1, thus all nodes transition to state START by time H_0. By (9.14), the lemma can be applied with $t = \Delta = H_0$, yielding times $p_{v,1}$, $v \in V_g$, satisfying the claimed skew bound of $2d$.

For the induction step from i to $i+1$, (9.15) yields that $v \in V_g$ transitions to READY no earlier than time

$$p_{v,i} + \frac{T_2}{\vartheta} \geq \max_{w \in V_g} \{p_{w,i}\} + \frac{T_2}{\vartheta} - 2d \geq \max_{w \in V_g} \{p_{w,i}\} + d$$

and no later than time

$$p_{v,i} + T_2 \leq \max_{w \in V_g} \{p_{w,i}\} + T_2 \,.$$

Thus, by (9.16) we can apply Lemma 9.15 with $t = \max_{w \in V_g} \{p_{w,i}\} + T_2$ and $\Delta = \left(1 - \frac{1}{\vartheta}\right) T_2 + 2d$, yielding pulse times $p_{v,i+1}$, $v \in V_g$, satisfying the stated skew bound.

It remains to show that $\min_{v \in V_g} \{p_{v,i+1}\} - \max_{v \in V_g} \{p_{v,i}\} \geq \frac{T_2+T_3}{\vartheta} - 2d$ and $\max_{v \in V_g} \{p_{v,i+1}\} - \min_{v \in V_g} \{p_{v,i}\} \leq T_2 + T_3 + 3d$. By Lemma 9.15,

$$\begin{aligned} p_{v,i+1} &\in \left(t - \Delta + \frac{T_3}{\vartheta}, t + T_3 + d\right) \\ &= \left(\max_{w \in V_g} \{p_{w,i}\} + \frac{T_2 + T_3}{\vartheta} - 2d, \max_{w \in V_g} \{p_{w,i}\} + T_2 + T_3 + d\right). \end{aligned}$$

Thus, the first bound is satisfied. The second follows as well, as we have already shown that $\max_{w \in V_g} \{p_{w,i}\} \leq \min_{w \in V_g} \{p_{w,i}\} + 2d$. □

Theorem 9.17. *Suppose $3f < n$, $H_v(0) \in [0, H_0)$ for all $v \in V$ and some known $H_0 \in \mathbb{R}^+$, and choose any $T \geq 3\vartheta d$. Then we can solve the pulse synchronization problem with $\mathcal{S} = 2d$, $P_{\min} = T$, and $P_{\max} = \vartheta T + (5 + 2(\vartheta - 1))d$, where each node generates its first pulse by time $H_0 + (\vartheta - 1)T + (3 + 2(\vartheta - 1))d$.*

Proof. Set $T_1 := \vartheta H_0$, $T_2 := T$, and $T_3 := (\vartheta - 1)T + 2\vartheta d$. By the assumption that $H_0 > H_v(0)$ for all $v \in V_g$, these choices satisfy Equations (9.13) to (9.16). We apply Lemma 9.16 to solve pulse synchronization with the stated skew and period bounds. As all correct nodes switch to PROPOSE for the first time by local (and thus also real) time $H_0 + T_3$, the first pulse is generated at each correct node by time $H_0 + T_3 + d = H_0 + (\vartheta - 1)T + (3 + 2(\vartheta - 1))d$. □

We remark that, by making $T_2 + T_3$ large, the ratio $P_{\max}/P_{\min}$ can be brought arbitrarily close to ϑ. On the other hand, we can go for the minimal choice $T_2 = 3\vartheta d$ and $T_3 = (3\vartheta^2 - \vartheta)d$, yielding $P_{\min} = 3\vartheta d$ and $P_{\max} = (3\vartheta^2 + 2\vartheta + 2)d$.

E9.17 Derive a clock synchronization algorithm using Theorem 9.11. Do the results so far leave room for improvement in some of the guarantees?

9.6 Implementing the Srikanth-Toueg Algorithm

The state machine representation in Figure 9.4 is already a good starting point for a hardware implementation of the algorithm. In the following we will do a top-down approach by

1. constructing the state machine, assuming the guards $G1$ to $G5$ are available as digital signals,
2. building the guards, assuming binary signals about the expiration status of the timers $\langle T_1 \rangle$ to $\langle T_3 \rangle$, about $H_v(t)$, as well as about the conditions C_G ($< f$) and C_{GE} ($\geq n - f$) of the propose flags are available,
3. building the timers,
4. deriving the conditions C_G and C_{GE} from the propose flags PF_i, and finally
5. maintaining the propose flags.

The block diagram shown in Figure 9.5 gives the top level view of the state machine design.

For the moment we assume that we build our state machine in a fully synchronous environment where a suitable clock HW_CLK is available for clocking the registers and the timers.

According to the algorithm, the communication between the instances of the state machine is established through the propose flags, namely by having each state machine broadcast its generated propose flag Pi to all others (plus itself). Note that these instances will have different clocks supplying them – otherwise there would be no need for a (pulse) synchronization. Our implementation of the "broadcast" will be a set of n signal lines, each driven by one instance of the algorithm ("node") and being routed to the inputs of all nodes. Should one of these signal lines become defective, we may see asymmetric faults, where some of the receiving nodes are cut off, while others are not. This can be mapped to a Byzantine behavior of the sender (or, alternatively, the receiver) and is hence well within our fault model. Another way of experiencing Byzantine behavior would be a node driving a voltage level on its associated line that is in between HI and LO. Then some receivers may regard this as a HI, and others as LO.

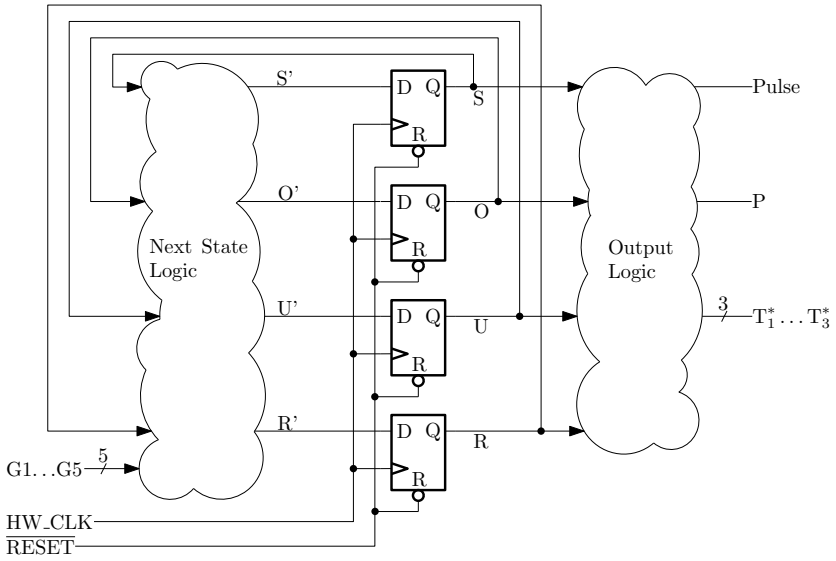

Figure 9.5

Block diagram of the state machine implementing the Srikanth-Toueg algorithm. the flip flops in the center hold the current state, encoded in the signals S, O, U, and R. The block "Next State logic" at the left computes, based on current state and input, the values (S', O', U', R') that shall be captured by the flip flops upon the next active clock edge. The "Output Logic" at the right derives, according to the current state, the output signals $Pulse$ and P, as well as the reset signals T_i* for the timers that will be required later on.

Worse, unless we protect the receiver from this, their logic may be affected by metastability, cf. **??**.

9.6.1 State Machine Design

To cover the space of 5 states that our state machine uses we need at least 3 bits. If we are slightly more generous and invest 4 bits, we can obtain a very intuitive state representation that will pay off later on, as follows: For the active states we use a one-hot encoding with

- bit S being active in state START,
- bit O in state PROPOSE,
- bit U in state PULSE, and
- bit R in state READY.
- For state RESET we conveniently use $(SOUR) = (0000)$, so we can (asynchronously) clear all state registers upon reset.

Next we have to determine how to move forward from one state to a next. This is particularly simple for the given state machine, as each state has one single clear successor state – the guards just determine the *moment* when a transition occurs, but have no influence on the *selection* of the next state that is attained. Consequently (and thanks to our investment into a one-hot encoding), our next state logic becomes very simple, see Figure 9.6.

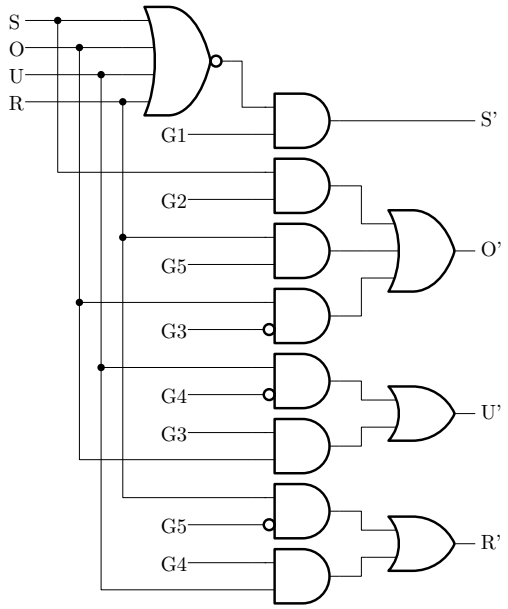

Figure 9.6
Next State Logic for the state machine. The OR gates make sure that the state is assumed when the state machine is in the predecessor state and the guard is valid (decoded at the AND with no inverted inputs), and that it is, once active, held active until the guard for the next state becomes valid (AND with the inverted input).

The outputs produced by our state machine can be easily derived as well: The propose signal needs to be sent when the state machine enters state PROPOSE, so we get PF = O'. Similarly, for the pulse we get PULSE = U. The use of the $Ti*$ signals for triggering the timers will be explained below.

In our synchronous implementation we can be sure that each state is maintained at least for one clock period. For PULSE this is not important, as this state is not left anyway before timer T_2 expires. However, PROPOSE is visited just to see if a sufficient number of propose flags has been set, and hence may indeed be visited for just one period of HW_CLK. In that case this period must be

long enough for the resulting propose pulse to be safely recognized by all other nodes. If this is not guaranteed, a suitable number C_{min} of clock states must be spent in PROPOSE. This can be enforced by a counter that starts counting when PROPOSE is entered and whose count reaching C_{min} forms an additional condition for moving to the next state.

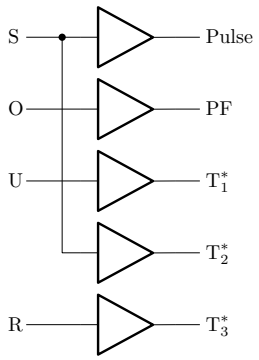

Figure 9.7
Output Logic for the state machine. Due to the chosen one-hot encoding of the states it is easy to derive the desired output signals. The buffers could be omitted, they are inserted here to formally separate the signals with different names at their inputs and outputs.

E9.18 With 4 bits representing the state, our pipeline could actually have up to 16 states. What happens when the pipeline, as a consequence of a bit flip, gets into one of the unused states? Could we extend the combinational parts of the state machine (next-state logic and output logic) to make it more robust?

9.6.2 Deriving the Guards

For deriving the guard signals Gx we can directly apply the table given in Figure 9.4: So, unsurprisingly, we get the circuit shown in Figure 9.8.

9.6.3 Building the Timers

Assuming that we have a local crystal clock available, it is easy to transform the time-outs required by the algorithm into equivalent counter values Ci. Once we need to trigger a given time-out Ti, we can reset the counter to zero and wait until the count matches that Ci. While comparison with an arbitrary, changing pattern is somewhat expensive to implement, here we fortunately have a constant Ci. So we may as well pre-load the counter to Ci and have it count down; then checking for count 0 becomes as simple as a NOR over

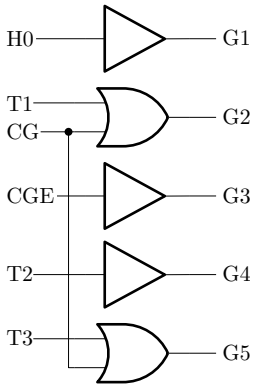

Figure 9.8
Derivation of the guard signals. This circuit logically implements the conditions stated in Figure 9.4

all bits. The pre-loading can be done by presetting or clearing the counter's individual flip flops upon trigger. Still we need a trigger signal $Tx*$ for this purpose. Here again the one-hot coding of our state machine proves beneficial: Assuming that (as usual) register clear and preset are low-active, we can have $T1* = S$, $T2* = U$, and $T3* = R$, as shown in Figure 9.7. With this assignment the timer required for the guard to leave START / PULSE / READY is triggered exactly when the respective state is entered. More precisely, the respective clear and preset signals of the individual registers are kept active all the time and only released when that state is entered and the counter is supposed to count down. In order to avoid metastability issues at the interfaces, it is highly advisable to use the same clock source HW_CLK for counter and state machine. A simple example for a timer is shown in Figure 9.8: Here the timer is kept initialized to 100 until $Ti*$ is released (carefully note the connection of $Ti*$ to the preset (S) and clear (R) inputs of the individual flip flops. Consequently, the next four clock edges will make it count down to zero (following the sequence 100,011,010,001,000). Then, on the next clock edge the state machine will sense the 1 on Ti. So overall we have implemented a time-out of 5 clock cycles.

E9.19 The combinational logic in a binary counter is relatively expensive. Think about which property of the counter is actually required and whether other, maybe cheaper, types of counters could do the job.

E9.20 Checking the counter for all zero is a simple way of detecting its expiration. What happens if, due to a glitch, we miss the zero-detection in the very cycle when

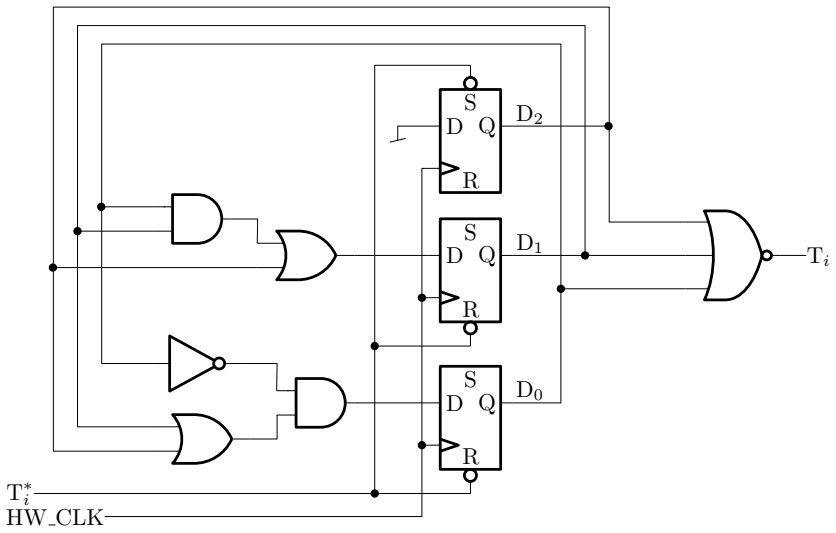

Figure 9.9
Example implementation of a time-out. A 3-bit counter is initialized to 100 by pulling $Ti*$ to 0. Upon release of $Ti*$, a count down sequence starts. As soon as count 000 is reached, Ti is raised.

the counter reaches that value? What will a counter normally do? What will the signal Ti do? What would be required for robust solution? How does the counter shown in Figure 9.9 perform in this respect?

Checking for $H_v(t) = H_0$ is a different problem as it just involves checking the match of the local time with a given threshold value. So there is no need to establish a time reference through a counter here. However, as the local time is not under our control here, we obviously cannot use the approach of counting down. So we need to do a more complicated comparison, as shown in Figure 9.10. In a naive approach we could use an AND gate with the number of inputs matching the bit width of H_0 and put an inversion at those inputs where H_0 holds a 0. In modern CMOS technologies, however, AND gates with more than 4 inputs (sometimes even 3) do not reliably work, so larger ones are built by cascading the available smaller ones, as shown in the figure.

9.6.4 Evaluating the Propose Flags

To generate the status signal CG we need to check whether among n inputs (the propose flags PF_i for all nodes) we have more than f activated. This so called "threshold function" is more costly than it initially seems. For the example of $f = 1$ and $n = 4$ we need at least two inputs activated, so the input patterns

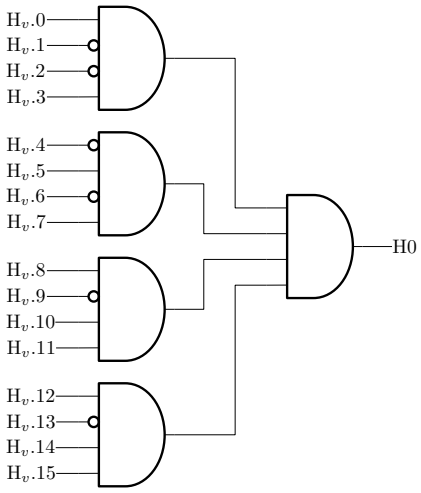

Figure 9.10
Example implementation of a time comparison $H_v(t) = H_0$ for the example of $H_0 = 1101110110101001$. A 16 bit wide AND is composed by cascading four 4 bit wide AND gates. All inputs where the corresponding bit position in H_0 holds a 0 are inverted.

(0011), (0101), (0110), (0111), (1001), (1010), (1011), (1100), (1101), (1110), (1111)

shall activate the CG output, while the others shall not.

Writing a truth table for this function is straightforward and will allow a sum-of-products implementation, possibly with a subsequent optimization. Already here it becomes apparent, that this approach does not scale well; in fact the number of terms that cause the output to be activated scales with $\binom{n}{f+1}$, and there is not much optimization possible. However, there are hardly any better solutions available. Sorting the inputs such that all ones are at the bottom bits of the output word, and all zeros in the upper positions, and then checking the appropriate bit position (in our example the second one from the bottom) for a one, is an alternative. The associated cost of $O(n \log(f + 1))$ is exponentially cheaper than the naive approach, but still becomes painful for larger f. Asymptotically yet cheaper is a (binary) adder tree. It has size $\Theta(n)$, because the number of nodes per level drops exponentially in the distance from leaves. As we can add in binary, the size of adders grows only logarithmically with distance from leaves. However, the sorting network should be (asymptotically) faster, because the comparators consist only of two gates, while the adders will have depth $\Theta(\log \log f)$. For the example of $f = 1$ and $n = 4$ Figure 9.11 shows an implementation for deriving CG.

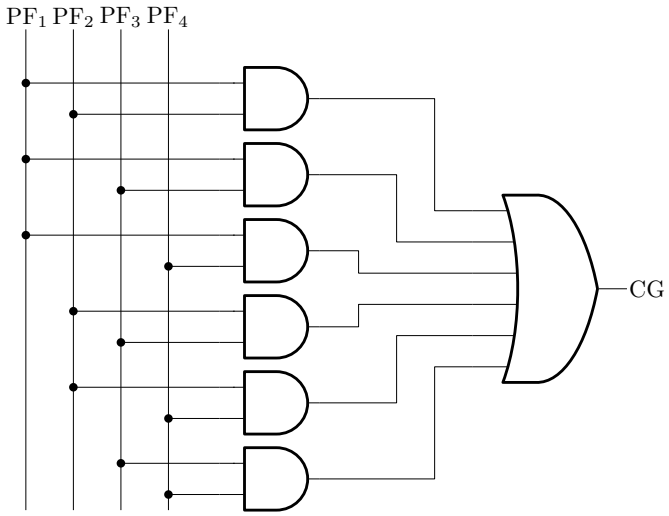

Figure 9.11
Implementation of the threshold gate for generating CG for the example of $n = 4$ and $f = 1$, illustrating, how all combinations of bit pairs need to be explicitly enumerated. Therefore, this approach scales badly for larger n, and other implementations based on sorting networks or adder trees should be considered.

In the same way the status signal CGE can be generated to indicate when condition $\geq n - f$ is fulfilled.

E9.21 The problem of checking whether at least $n-f$ inputs are activated (as required for CGE) is complementary to the one of checking whether more than f inputs are activated (as for CG): In fact, the former can be rephrased to checking whether more than f inputs are *not* activated. Using this insight, think about how to modify the circuit that generates CG to make it applicable for solving CGE.

9.6.5 Maintaining the Propose Flags

Once a node i activates its propose signal P_i, all receiving nodes must set the corresponding local flag PF_i and keep it set until the local state machine clears it – even if the sending node de-activates P_i later on. This "sticky bit" can be implemented using a D-flip flop whose data input is a logical OR of the incoming P_i and the same flip flop's output. The reset of the flag would then be an (asynchronous) register clear (low active), accomplished when any of the states START / PROPOSE / READY is entered. This simple solution is shown in the left part of Figure 9.12.

There is, however, one issue that calls for a more elaborate implementation: The algorithm assumes the PF_i to be strictly monotonic during most phases –

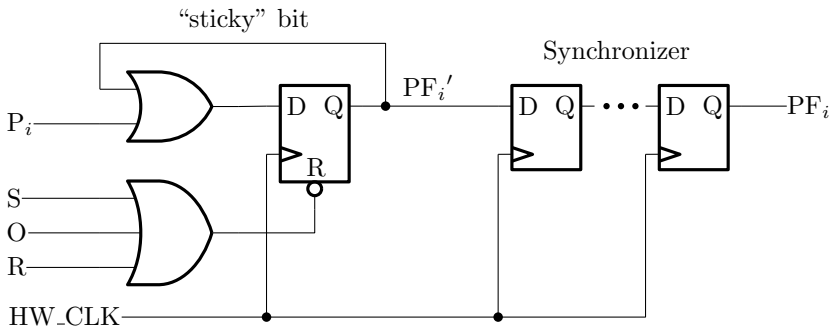

Figure 9.12
Implementation of the "sticky bit" required to maintain the propose flag asynchronously received from an other node. The synchronizer on the right mitigates the risk of metastability to propagate down to PF_i

once set they would never go to zero before being reset by the algorithm. This assumption is somewhat threatened by the potential for metastability that we experience at this sticky register: Recall that the other nodes work in different clock domains, so the activation of the P_i occurs uncorrelated with the local clock that drives the register. Therefore metastability cannot be avoided, even without any node failing, when P_i is sampled right at its rising transition. Even if one can assume that P_i will remain activated for more than one clock cycle and so a stable logic 1 is guaranteed to be sampled next, the first, metastable cycle may produce a glitch (shorter than one clock cycle though). A subsequent synchronizer circuit buys the metastable bit more time to resolve and hence mitigates that risk. So to create a glitch at the output, the metastability would have to persist until the last register stage. The probability for such a case can be calculated and, by making the chain sufficiently long, be made arbitrarily small.

In the same way, a faulty node may activate its P_i only shortly and at the worst possible moment to make the register chain metastable. Again, however, this has only an effect if the metastable upset propagates through the whole chain.

Another aspect worth noting is that the state diagram in Figure 9.4 suggests to clear the propose flags upon a state *transition*. The transition to READY, e.g., can be safely detected by a flip flop that captures the value of R into a signal R_{prev}. Then the case when $R = 1$ and $R_{prev} = 0$ indicates a rising edge on R for one clock cycle and can hence be conveniently decoded into the required clear pulse for the propose flags. As we have to do this in two places, we need two extra registers.

However, a closer look at Figure 9.4 reveals that the propose flags are not relevant in states RESET and PULSE anyway, so the propose flags can be cleared already *during* these states (and not just upon leaving them). This simplifies the clear condition for the propose flags to an OR of $(SOUR) = (0000)$ and $(U = 1)$, which yields $(SOR) = (000)$. This is easy to implement with a NOR and does not require a flip flop. However, this implementation formally does not follow the given algorithm.