Contents

Lecture 1

Synchronizing Clocks

In this lecture series, we are going to approach fault-tolerant clock generation and distribution from a theoretical angle. This means we will formalize parametrized problems and prove impossibilities, lower bounds, and upper bounds for them. However, make no mistake: these tasks are derived from real-world challenges, and a lot of the ideas and concepts can be used in the design of highly reliable and scalable hardware solutions. The first lecture focuses on the basic task at hand, without bells and whistles. Asking more refined questions will prompt more refined answers later in the course; nonetheless, the initial lecture offers a good impression of the general approach and flair of the course.

1.1 The Clock Synchronization Problem

We describe a distributed system by a simple, connected graph $G = (V, E)$ (see Appendix A), where V is the set of $n := |V|$ nodes (our computational entities, e.g., computers in a network) and nodes v and w can directly communicate if and only if there is an edge $\{v, w\} \in E$. Each node is equipped with a *local* or *hardware clock.* We model this clock as a strictly increasing function $H_v \colon \mathbb{R}_0^+ \to \mathbb{R}_0^+$, whose rate of increase is between 1 and $\vartheta > 1$:

$$\forall v \in V, t, t' \in \mathbb{R}_0^+, t \geq t' \colon t - t' \leq H_v(t) - H_v(t') \leq \vartheta(t - t')\,,$$

where $t \in \mathbb{R}_0^+$ denotes “perfect” Newtonian *real time* (which is not known to the nodes). For simplicity, we assume that hardware clocks are differentiable and denote the derivative by h_v. Thus, the above inequalities are equivalent to $h_v(t) \in [1, \vartheta]$ at all times t. However, all claims can be shown solely based on the above requirement.

Note that even if the hardware clocks of nodes v and w would be initially perfectly synchronized (i.e., $H_v(0) = H_w(0)$), over time they could drift apart at a rate of up to $\vartheta - 1$. Accordingly, we refer to $\vartheta - 1$ as the *maximum drift,* or, in short, *drift.* In order to establish or maintain synchronization, nodes need to communicate with each other. To this end, on any edge $\{v, w\}$, v can send messages to w (and vice versa). However, it is not known how long such a message is under way. A message sent at time t is received at a time $t' \in (t + d - u, t + d)$, where d is the (maximum) *delay* and u is the (delay) *uncertainty.* We subsume possible delays due to computations in d, i.e., at the time t' when the message is received in our abstract model, all updates to the state of the receiving node take effect and any message it sends in immediate response is sent. Nodes may also send messages later, at a time t'' specified by some hardware clock value $H > H_v(t')$; the messages are then sent at the time t'' when $H_v(t'') = H$, unless reception of a message at an earlier time makes v "change its mind."

An *execution* of an algorithm on a system is given by specifying clock functions H_v as above to each $v \in V$ and assigning to each message a reception time $t' \in (t + d - u, t + d)$, where t is the time it was sent. Note that by performing this inductively over increasing reception times enables to always determine from the execution up to the current time what the state of each node is and which messages are in transit, i.e., choosing clock functions and delays fully determines an execution.

The *clock synchronization problem* requires each node $v \in V$ to compute a *logical clock* $L_v \colon \mathbb{R}_0^+ \to \mathbb{R}_0^+$, where $L_v(t)$ is determined from the current state of the node (computed when receiving the most recent message, or the initial state if no message has been received yet) and $H_v(t)$. The goal is to minimize, for any possible execution $\mathcal{E}$, the *global skew*

$$\mathcal{G} := \sup_{t \in \mathbb{R}_0^+} \{\mathcal{G}(t)\},$$

where

$$\mathcal{G}(t) := \max_{v,w \in V} \{|L_v(t) - L_w(t)|\} = \max_{v \in V}\{L_v(t)\} - \min_{v \in V}\{L_v(t)\}$$

is the *global skew at time t.*

For simplicity, this notation does not reflect the dependence on the execution. The goal is to bound $\mathcal{G}$ for all possible executions, yet frequently we will argue about specific executions. We will make the dependence explicit only when reasoning about different executions concurrently.

Remarks:

- For practical purposes, clocks are discrete and bounded (i.e., wrap around to 0 after reaching a maximum value), and nodes may not be able to read them (perform computations, send messages, etc.) at arbitrary times. We hide these issues in our abstraction, as they can be handled easily, by adjusting d and u to account for them and making minor adjustments to algorithms.

- A cheap quartz oscillator has a drift of $\vartheta - 1 \approx 10^{-5}$, which will be more than accurate enough for running all the algorithms we'll get to see. In some cases, however, one might only want to use basic digital ring oscillators (an odd number of inverters arranged in a cycle), for which $\vartheta - 1 \approx 10\%$ is not unusual.

- There are other forms of communication than point-to-point message passing. Changing the mode of communication has, in most cases, little influence on a conceptual level, though.

- Another issue is that clocks may not be perfectly synchronized at time 0. After all, we want to run a synchronization algorithm to make clocks agree, so assuming that this is already true from the start would create a chicken-and-egg problem. But if we assume that initial clock values are arbitrary, we cannot bound $\mathcal{G}$. Instead, we assume that, for some $F \in \mathbb{R}^+$, it holds that $H_v(0) \in [0, F]$ for all $v \in V$. We then can bound $\mathcal{G}$ in terms of F (and, of course, other parameters).

- In order to perform induction over message sending and/or reception times, we need the additional assumption that nodes send only finitely many messages in finite time. As physics ensure that is the case (and any reasonable algorithm should not attempt otherwise), we implicitly make this assumption throughout the course.

1.2 The Max Algorithm

Let's start with our first algorithm. It's straightforward: Nodes initialize their logical clocks to their initial hardware clock value, increase it at the rate of the hardware clock, and set it to the largest value they can be sure that some other node has reached. To make the latter useful, each node broadcasts its clock value (i.e., sends it to all neighbors) whenever it reaches an integer multiple of some parameter T. See Algorithm 1.1 for the pseudocode.

Algorithm 1.1: Basic Max Algorithm. Parameter $T \in \mathbb{R}^+$ controls the message frequency. The code lists the actions of node v at time t.

```
$L_v(0) := H_v(0)$
at all times, increase $L_v$ at the rate of $H_v$
if received $\langle L \rangle$ at time $t$ and $L > L_v(t)$ then
    $L_v(t) := L$
if $L_v(t) = kT$ for some $k \in \mathbb{N}$ then
    send $\langle L_v(t) \rangle$ to all neighbors
```

Lemma 1.1. *In a system executing Algorithm 1.1, it holds that*

$$\mathcal{G}(t) \leq \vartheta d D + (\vartheta - 1)T$$

for all $t \geq dD + T$, where D is the diameter of G.

Proof. Set $L := \max_{v \in V}\{L_v(t - dD - T)\}$. No node ever sets its logical clock to a value that has not been reached by another node before. Together with the fact that hardware clocks increase at rate at most ϑ, this implies that

$$\max_{v \in V}\{L_v(t)\} \leq \max_{v \in V}\{L_v(t - dD - T)\} + \vartheta(dD + T) = L + \vartheta(dD + T)\,.$$

Let v be a node such that $L_v(t - dD - T) = \max_{w \in V}\{L_w(t - dD - T)\}$. As the logical clock of v increases at least at rate 1, the minimum rate of its hardware clock, and is never set back to a smaller value, we have that $L_v(t') = kT$ for some $k \in \mathbb{N}$ and $t' \in [t, t + T)$. At time t', v sends $\langle kT \rangle = \langle L_v(t') \rangle$ to all neighbors. These will receive it before time $t' + d$ and, if they have not reached clock value kT and sent a message $\langle kT \rangle$ yet, do so now. By induction, every node within D hops of v will receive a message $\langle kT\rangle$ by time $t'+dD$. As we assume G to be connected, these are all nodes.

Consider any node $w \in V$. As w sets L_w to value kT when receiving a message $\langle kT\rangle$ (unless it is already larger), we have that

$$\begin{aligned}L_w(t) &\geq L_w(t'+dD)+t-(t'+dD)\\ &\geq L_v(t')+t-(t'+dD)\\ &\geq L_v(t-dD-T)+t'-(t-dD-T)+t-(t'+dD)=L+T\,.\end{aligned}$$

As w is arbitary, it follows that

$$\mathcal{G}(t)=\max_{v\in V}\{L_v(t)\}-\min_{w\in V}\{L_w(t)\}\leq \vartheta dD+(\vartheta-1)T\,. \qquad \square$$

Theorem 1.2. *Set $H := \max_{v\in V}\{H_v(0)\}-\min_{v\in V}\{H_v(0)\}$. Then Algorithm 1.1 achieves*

$$\mathcal{G}\leq \max\{H,dD\}+(\vartheta-1)(dD+T)\,.$$

Proof. Consider $t\in\mathbb{R}_0^+$. If $t\geq dD+T$, then $\mathcal{G}(t)\leq dD+(\vartheta-1)T$ by Lemma 1.8. If $t<dD+T$, then for any $v,w\in V$ we have that

$$L_v(t)-L_w(t)\leq L_v(0)-L_w(0)+(\vartheta-1)t\leq H+(\vartheta-1)(dD+T)\,. \qquad \square$$

Remarks:

- H reflects the skew on initialization. Getting H small may or may not be relevant to applications, but it yields little understanding of the overall problem; hence we neglect this issue here.
- Making H part of the bound means that we do not bound $\mathcal{G}$ for all executions, as the model allows for executions with arbitrarily large initial clock offsets $H_v(0)-H_w(0)$. An unconditional bound will require to ensure that H is small — but of course this "unconditional" bound then still relies on the assumptions of the model.
- Is this algorithm good? May it even be optimal in some sense?

1.3 Lower Bound on the Global Skew

To argue that we performed well, we need to show that we could not have done (much) better (in the worst case). We will use the *shifting technique,* which enables to "hide" skew from the nodes. That is, we construct two executions which look completely identical from the perspective of all nodes, but different hardware clock values are reached at different times. No matter how the algorithm assigns logical clock values, in one of the executions the skew must be large — provided that nodes *do* increase their clocks. First, we need to state what it means that two executions are *indistinguishable* at a node.

Definition 1.3 (Indistinguishable Executions)**.** *Executions $\mathcal{E}_0$ and $\mathcal{E}_1$ are* indistinguishable at node $v\in V$ until local time H, *if $H_v^{(\mathcal{E}_0)}(0)=H_v^{(\mathcal{E}_1)}(0)$ (where the superscripts indicate the execution) and, for $i\in\{0,1\}$, for each message v*

receives at local time $H' \leq H$ in $\mathcal{E}_i$ from some neighbor $w \in V$, it receives an identical message from w at local time H' in $\mathcal{E}_{1-i}$. If we drop the "until local time H," this means that the statement holds for all H, and if we drop the "at node v," the statement holds for all nodes.

Remarks:

- If two executions are indistinguishable until local time H at $v \in V$, it sends the same messages in both executions and computes the same logical clock values — in terms of its *local* time — until local time H. This holds because our algorithms are deterministic and all actions nodes take are determined by their local perception of time and which messages they received (and when).

- As long as we can ensure that the receiver of each message receives it at the same *local* time in two executions without violating the constraint that messages are under way between $d-u$ and d *real* time in both executions, we can inductively maintain indistinguishability: as long as this condition is never violated, each node will send the same messages in both executions at the same hardware times.

Before showing that we cannot avoid a certain global skew, we need to add a requirement, namely that clocks actually behave like clocks and make progress. Note that, without such a constraint, setting $L_v(t) = 0$ at all $v \in V$ and times t is a "perfect" solution for the clock synchronization problem.

Definition 1.4 (Amortized Minimum Progress). *For $\alpha \in \mathbb{R}^+$, an algorithm satisfies the* amortized α-progress condition, *if there is some $C \in \mathbb{R}_0^+$ such that $\min_{v \in V}\{L_v(t)\} \geq \alpha t - C$ for all $t \in \mathbb{R}_0^+$ and all executions.*

We now prove that we cannot only "hide hardware clock skew," but also keep nodes from figuring out that they might be able to advance their logical clocks slower than their hardware clocks in such executions.

Lemma 1.5. *Fix an arbitrary algorithm and any node $v \in V$. For arbitrarily small $\varepsilon > 0$, there are executions $\mathcal{E}_v$ and $\mathcal{E}_1$ that are indistinguishable such that*

- $H_x^{(\mathcal{E}_1)}(t) = t$ *for all $x \in V$ and t,*

- $H_v^{(\mathcal{E}_v)}(t) = H_v^{(\mathcal{E}_1)}(t) + uD - \varepsilon$ *for all $t \geq t_0 := \frac{uD-\varepsilon}{\rho-1}$ for some $\rho \in (1, \vartheta]$,*

- $H_w^{(\mathcal{E}_v)}(t) = t$ *for some $w \in V$ and all t.*

Proof. In both executions and for all $x \in V$, we set $H_x(0) := 0$. Denote by $d(x,y)$ the distance (i.e., hop count of a shortest path) between nodes x and y, and fix some node $w \in V$ with $d(v,w) = D$. Abbreviate $d(x) := d(x,w) - d(x,v)$. Execution $\mathcal{E}_1$ is given by running the algorithm with all hardware clock rates being 1 at all times and the message delay from x to y being $d - (\frac{1}{2} - \frac{d(x)-d(y)}{4})u$.

Observe that $d(x) \in [-D, D]$, where $d(v) = D$ and $d(w) = -D$, and that $d(\cdot)$ differs by at most 2 between neighbors. In $\mathcal{E}_v$, we set the hardware clock rate of node $x \in V$ to $1 + \frac{(\rho-1)(d(x)+D)}{2D}$ at all times $t \leq t_0$ and 1 at all times $t > t_0$ (we will specify $\rho \in (1, \vartheta)$ later). This implies that

$$H_v^{(\mathcal{E}_v)}(t_0) = \rho t_0 = t_0 + (\rho-1)t_0 = t_0 + uD - \varepsilon = H_v^{(\mathcal{E}_1)}(t_0) + uD - \varepsilon \quad \text{and}$$
$$H_w^{(\mathcal{E}_v)}(t_0) = t_0 \,.$$

As clock rates are 1 from time t_0 on, this means that the hardware clocks satisfy all stated constraints.

It remains to specify message delays and show that the two executions are indistinguishable. We achieve this by simply ruling that a message sent from some $x \in V$ to a neighbor $y \in V$ in $\mathcal{E}_v$ arrives at the same local time at y as it does in $\mathcal{E}_1$. By induction over the arrival sending times of messages, then indeed all nodes also send identical messages at identical local times in both executions, i.e., the executions remain indistinguishable at all nodes and times. However, it remains to prove that this results in all message delays being in the range $(d-u, d)$.

To see this, recall that for any $\{x, y\} \in E$, we have that $|d(x) - d(y)| \leq 2$. As clock rates are 1 after time t_0 and constant before, and all hardware clocks are 0 at time 0, the maximum difference between any two local times between neighbors is attained at time t_0. We compute

$$H_x^{(\mathcal{E}_v)}(t_0) - H_y^{(\mathcal{E}_v)}(t_0) = \frac{d(y)-d(x)}{2D} \cdot (\rho-1)t_0 = \frac{d(y)-d(x)}{2} \cdot \left(u - \frac{\varepsilon}{D}\right) .$$

In execution $\mathcal{E}_1$, a message sent from x to y at local time $H_x^{(\mathcal{E}_1)}(t) = t$ is received at local time $H_y^{(\mathcal{E}_1)}(t) = H_x^{(\mathcal{E}_1)}(t) + d - (\frac{1}{2} - \frac{d(x)-d(y)}{4})u$. If a message is sent at time t in $\mathcal{E}_v$, we have that

$$\begin{aligned}
&H_y^{(\mathcal{E}_v)}(t+d)\\
&\geq H_y^{(\mathcal{E}_v)}(t) + d\\
&= H_x^{(\mathcal{E}_v)}(t) + d + \frac{d(x)-d(y)}{2} \cdot \left(u - \frac{\varepsilon}{D}\right)\\
&= H_x^{(\mathcal{E}_v)}(t) + d - \left(\frac{1}{2} - \frac{d(x)-d(y)}{4}\right)u + \frac{2+d(x)-d(y)}{4} \cdot u - \frac{(d(x)-d(y))\varepsilon}{2D}\\
&> H_x^{(\mathcal{E}_v)}(t) + d - \left(\frac{1}{2} - \frac{d(x)-d(y)}{4}\right)u
\end{aligned}$$

where the last inequality uses that $d(x) - d(y) \geq -2$ and assumes that $\varepsilon < uD$, i.e., ε is sufficiently small. On the other hand, as clock rates in $\mathcal{E}_v$ are at most ρ,

$$\begin{aligned}
&H_y^{(\mathcal{E}_v)}(t+d-u)\\
&\leq H_y^{(\mathcal{E}_v)}(t) + \rho d - u\\
&= H_x^{(\mathcal{E}_v)}(t) + \rho d - u + \frac{d(x)-d(y)}{2} \cdot \left(u - \frac{\varepsilon}{D}\right)\\
&= H_x^{(\mathcal{E}_v)}(t) + \rho d - \left(\frac{1}{2} - \frac{d(x)-d(y)}{4}\right)u + \frac{d(x)-d(y)-2}{4}u - \frac{(d(x)-d(y))\varepsilon}{2D} .
\end{aligned}$$

We want to bound this term by $H_x^{(\mathcal{E}_v)}(t) + d - \left(\frac{1}{2} - \frac{d(x)-d(y)}{4}\right)u$, which is equivalent to requiring that

$$(\rho-1)d + \frac{d(x)-d(y)-2}{4} \cdot u - \frac{(d(x)-d(y))\varepsilon}{2D} < 0 .$$

We are still free to choose ρ from $(1, \vartheta]$. We set $\rho := \min\{1 + \varepsilon/(2dD), \vartheta\}$, implying that the left hand side is smaller than 0 if $d(x) - d(y) = 2$. The other case is that $d(x) - d(y) \leq 1$, and choosing ε (and thus also $\rho - 1$) sufficiently close to 0 ensures that the inequality holds. □

Theorem 1.6. *If an algorithm satisfies the amortized α-progress condition for some $\alpha \in \mathbb{R}^+$, then $\mathcal{G} \geq \frac{\alpha u D}{2}$, even if we are guaranteed that $H_v(0) = 0$ for all $v \in V$.*

Proof. From Lemma 1.5, for arbitrarily small $\varepsilon > 0$ we have two indistinguishable executions $\mathcal{E}_v$, $\mathcal{E}_1$ and nodes $v, w \in V$ such that

- $H_v^{(\mathcal{E}_1)}(t) = H_w^{(\mathcal{E}_1)}(t) = H_w^{(\mathcal{E}_v)}(t) = t$ for all $t \in \mathbb{R}_0^+$ and
- there is a time t_0 such that $H_v^{(\mathcal{E}_v)}(t) = t + uD - \varepsilon$ for all $t \geq t_0$.

Because the algorithm satisfies the amortized α-progress condition, we have that $L_v^{(\mathcal{E}_1)}(t) \geq \alpha t - C$ for all t and some $C \in \mathbb{R}_0^+$. We claim that there is some $t \geq t_0$ satisfying that

$$L_w^{(\mathcal{E}_1)}(t + uD - \varepsilon) - L_w^{(\mathcal{E}_1)}(t) \geq \alpha(uD - 2\varepsilon)\,. \tag{1.1}$$

Assuming for contradiction that this is false, set $\rho := \frac{\alpha(uD-2\varepsilon)}{uD-\varepsilon} < \alpha$ and consider times $t := t_0 + k(uD - \varepsilon)$ for $k \in \mathbb{N}$. We get that

$$L_w^{(\mathcal{E}_1)}(t) \leq L_w^{(\mathcal{E}_1)}(t_0) + \rho(t - t_0) = \alpha(t - t_0) - (\alpha - \rho)(t - t_0) + L_w^{(\mathcal{E}_1)}(t_0)\,.$$

Choosing $t > \frac{L_w^{(\mathcal{E}_1)}(t_0)+C}{\alpha-\rho}$, we get that $L_w^{(\mathcal{E}_1)}(t) < \alpha t - C$, violating the α-progress condition. Thus, we reach a contradiction, i.e., the claim must hold true.

Now let $t \geq t_0$ be such that (1.1) holds. As $H_w^{(\mathcal{E}_1)}(t) = H_w^{(\mathcal{E}_v)}(t)$, by indistinguishability of $\mathcal{E}_1$ and $\mathcal{E}_v$ we have that $L_w^{(\mathcal{E}_1)}(t) = L_w^{(\mathcal{E}_v)}(t)$. As $H_v^{(\mathcal{E}_1)}(t+uD-\varepsilon) = t + uD - \varepsilon = H_v^{(\mathcal{E}_v)}(t)$, we have that $L_v^{(\mathcal{E}_v)}(t) = L_v^{(\mathcal{E}_1)}(t + uD - \varepsilon)$. Hence,

$$\begin{aligned}
&L_w^{(\mathcal{E}_1)}(t + uD - \varepsilon) - L_v^{(\mathcal{E}_1)}(t + uD - \varepsilon)\\
&\geq L_w^{(\mathcal{E}_1)}(t) + \alpha(uD - 2\varepsilon) - L_v^{(\mathcal{E}_1)}(t + uD - \varepsilon)\\
&= L_w^{(\mathcal{E}_v)}(t) - L_v^{(\mathcal{E}_v)}(t) + \alpha(uD - 2\varepsilon)\,.
\end{aligned}$$

We conclude in at least one of the two executions, the logical clock difference between v and w reaches at least $\frac{\alpha u D}{2} - \varepsilon$. As $\varepsilon > 0$ can be chosen arbitrarily small, it follows that $\mathcal{G} \geq \frac{\alpha u D}{2}$, as claimed. □

Remarks:

- The good news: We have a lower bound on the skew that is linear in D. The bad news: typically $u \ll d$, so we might be able to do much better.
- When propagating information, we haven't factored in yet that we *know* that messages are under way for at least $d - u$ time. Let's exploit this!

Algorithm 1.2: Refined Max Algorithm.

```
$L_v(0) := H_v(0)$
at all times, increase $L_v$ at the rate of $H_v$
if received $\langle L \rangle$ at time $t$ and $L + d - u > L_v(t)$ then
    $L_v(t) := L + d - u$
if $H_v(t) = kT$ for some $k \in \mathbb{N}$ then
    send $\langle L_v(t) \rangle$ to all neighbors
```

1.4 Refining the Max Algorithm

Lemma 1.7. *In a system executing Algorithm 1.2, no $v \in V$ ever sets L_v to a value larger than $\max_{w \in V \setminus \{v\}}\{L_w(t)\}$.*

Proof. If any node $v \in V$ sends message $\langle L_v(t) \rangle$ at time t, it is not received before time $t + d - u$, for which it holds that

$$\max_{w \in V}\{L_w(t + d - u)\} \geq L_v(t + d - u) \geq L_v(t) + d - u\,,$$

as all nodes, in particular v, increase their logical clocks at least at rate 1, the minimum rate of increase of their hardware clocks. □

Lemma 1.8. *In a system executing Algorithm 1.2, it holds that*

$$\mathcal{G}(t) \leq ((\vartheta - 1)(d + T) + u)D$$

for all $t \geq (d + T)D$, where D is the diameter of G.

Proof. Set $L := \max_{v \in V}\{L_v(t - (d + T)D)\}$. By Lemma 1.7 and the fact that hardware clocks increase at rate at most ϑ, we have that

$$\max_{v \in V}\{L_v(t)\} \leq \max_{v \in V}\{L_v(t - (d + T)D)\} + \vartheta(d + T)D = L + \vartheta(d + T)D\,.$$

Consider any node $w \in V$. We claim that $L_w(t) \geq L + (d + T - u)D$, which implies

$$\max_{v \in V}\{L_v(t)\} - L_w(t) \leq L + \vartheta(d+T)D - (L + (d+T-u)D) = ((\vartheta-1)(d+T)+u)D\,;$$

as w is arbitary, this yields the statement of the lemma.

It remains to show the claim. Let $v \in V$ be such that $L_v(t - (d + T)D) = L$. Denote by $(v_{D-h} = v, v_{D-h+1}, \ldots, v_D = w)$, where $h \leq D$, a shortest v-w-path. Define $t_i := t - (D - i)(d + T)$. We prove by induction over $i \in \{D - h, D - h + 1, \ldots, D\}$ that

$$L_{v_i}(t_i) \geq L + i(d + T - u)\,,$$

where the base case $i = D - h$ is readily verified by noting that

$$L_v(t_i) \geq L_v(t - (d + T)D) + t_i - (t - (d + T)D) = L + i(d + T)\,.$$

For the induction step from $i - 1 \in \{D - h, \ldots, D - 1\}$ to i, observe that v_{i-1} sends a message to v_i at some time $t_s \in (t_{i-1}, t_{i-1} + T]$, as its hardware clock increases by at least T in this time interval. This message is received by v_i at some time $t_r \in (t_s, t_s + d) \subseteq (t_{i-1}, t_{i-1} + d + T)$. Note that $t_{i-1} < t_s < t_r < t_i$. If necessary, v_i will increase its clock at time t_r, ensuring that

$$\begin{aligned} L_{v_i}(t_i) &\geq L_{v_i}(t_r) + t_i - t_r \\ &\geq L_{v_{i-1}}(t_s) + d - u + t_i - t_r \\ &\geq L_{v_{i-1}}(t_s) + t_i - t_s - u \\ &\geq L_{v_{i-1}}(t_{i-1}) + t_i - t_{i-1} - u \\ &= L_{v_{i-1}}(t_{i-1}) + d + T - u \\ &\geq L + i(d + T - u)\,, \end{aligned}$$

where the last step uses the induction hypothesis. This completes the induction. Inserting $i = D$ yields that $L_w(t) \geq L_{v_D}(t_D) = L + (d + T - u)D$, as claimed, completing the proof. □

Theorem 1.9. *Set* $H \coloneqq \max_{v\in V}\{H_v(0)\} - \min_{v\in V}\{H_v(0)\}$. *Then Algorithm 1.2 achieves*

$$\mathcal{G} \leq \max\{H, uD\} + (\vartheta - 1)(d + T)D\,.$$

Proof. Consider $t \in \mathbb{R}_0^+$. If $t \geq (d+T)D$, then $\mathcal{G}(t) \leq uD + (\vartheta - 1)(d+T)D$ by Lemma 1.8. If $t < (d + T)D$, then for any $v, w \in V$ we have that

$$L_v(t) - L_w(t) \leq L_v(0) - L_w(0) + (\vartheta - 1)t \leq H + (\vartheta - 1)(d + T)D\,. \qquad \square$$

Remarks:

- Note the change from using logical clock values to hardware clock values to decide when to send a message. The reason is that increasing received clock values to account for minimum delay pays off only if the increase is also forwarded in messages. However, sending a message every time the clock is set to a larger value might cause a lot of messages, as now different values than kT for some $k \in \mathbb{N}$ might be sent. The compromise presented here keeps the number of messages in check, but pays for it by exchanging the $(\vartheta - 1)T$ term in skew for $(\vartheta - 1)TD$.

- Choosing $T \in \Theta(d)$ means that nodes need to send messages roughly every d time, but in return $\mathcal{G} \in \max\{H, uD\} + \mathcal{O}((\vartheta - 1)dD)$. Reducing T further yields diminishing returns.

- Typically, $u \ll d$, but also $\vartheta - 1 \ll 1$. However, if $u \ll (\vartheta - 1)d$, one might consider to build a better clock by bouncing messages back and forth between pairs of nodes. Hence, this setting makes only sense if communication is expensive or unreliable, and in many cases one can expect uD to be the dominant term.

- In the exercises, you will show how to achieve a skew of $\mathcal{O}(uD + (\vartheta - 1)d)$.

- So we can say that the algorithm achieves asymptotically optimal global skew (in our model). The lower bound holds in the worst case, but we have shown that it applies to *any* graph. So, for deterministic guarantees, changing the network topology has no effect beyond influencing the diameter.

- We neglected important aspects like local skew and fault-tolerance, which will keep us busy during the remainder of the course.

1.5 Afterthought: Stronger Lower Bound

Both of our algorithms are actually much more restrained in terms of clock progress than just satisfying an amortized lower bound of 1 on the rates.

Definition 1.10 (Strong Envelope Condition)**.** *An algorithm satisfies the* strong envelope condition, *if at all times and for all nodes $v \in V$, it holds that* $\min_{w\in V}\{H_w(t)\} \leq L_v(t) \leq \max_{w\in V}\{H_w(t)\}$.

Corollary 1.11. *For any algorithm satisfying the strong envelope condition, it holds that $\mathcal{G} \geq uD$, even if we are guaranteed that $H_v(0) = 0$ for all $v \in V$.*

Proof. Apply Lemma 1.5 for some $v \in V$ and $\varepsilon > 0$. We have that $H_x^{\mathcal{E}_1}(t) = t$ for all $x \in V$. The strong envelope condition thus entails that $L_x^{\mathcal{E}_1}(t) = H_x^{\mathcal{E}_1}(t) = t$ for all x and t. As $\mathcal{E}_v$ is indistinguishable from $\mathcal{E}_1$, it follows that also $L_x^{\mathcal{E}_v}(t) = H_x^{\mathcal{E}_v}(t)$ for all x and t. In particular, there is some $w \in V$ such that

$$L_v^{\mathcal{E}_v}(t_0) - L_w^{\mathcal{E}_w}(t_0) = uD - \varepsilon\,.$$

As this holds for arbitrarily small $\varepsilon > 0$, we conclude that indeed $\mathcal{G} \geq uD$, as claimed. □

Remarks:

- Thus, in some sense the term uD in the skew bound is optimal.
- If one merely requires the weaker bound $t \leq L_v(t) \leq \max_{v\in V}\{H_v(0)\} + \vartheta t$, then a lower bound of $\frac{uD}{\vartheta}$ can be shown.
- Playing with such progress conditions is usually of limited relevance, as one cannot gain more than a factor of 2 — unless one is willing to simply slow down everything.

What to Take Home

- The shifting technique is an important source of lower bounds. We will see it again.
- If all that we're concerned with is the global skew and we have no faults, things are easy.
- There are other communication models, giving slightly different results. However, in a sense, our model satisfies the minimal requirements to be different from an asynchronous system (in which nodes have no meaningful sense of time): They can measure time with some accuracy, and messages cannot be delayed arbitrarily.
- The linear lower bound on the skew is highly resilient to model variations. If delays are distributed randomly and independently, a probabilistic analysis yields skews proportional to roughly $\sqrt{D}$, though (for most of the time). This is outside the scope of this lecture series.

Bibliographic Notes

The shifting technique was introduced by Lundelius and Lynch, who show that even if the system is fully connected, there are no faults, and there is no drift (i.e., $\vartheta = 1$), better synchronization than $\left(1 - \frac{1}{n}\right) u$ cannot be achieved [LL84]. Biaz and Lundelius Welch generalized the lower bound to arbitrary networks [BW01]. Note that Jennifer Lundelius and Jennifer Lundelius Welch are the same person — and the double name "Lundelius Welch" will be frequently cited as Welch (as "Lundelius" will be treated as a middle name, both by typesetting systems and people who don't know otherwise). I will stick to "Welch" as well, but for a different reason: "the Lynch-Lundelius-Welch algorithm" is a mouthful, and "the Lynch-Welch algorithm" rolls off the tongue much better (I hope that I'll be forgiven if she ever finds out!).

As far as I know, the max algorithm has been mentioned first in writing by Locher and Wattenhofer [LW06] — but not because it is such a good synchronization algorithm, but rather due its terrible performance when it comes to the skew between neighboring nodes (see excersise). Being an extremely straightforward solution, it is likely to appear earlier and in other places and should be considered folklore. In contrast to the earlier works mentioned above (and many more), [LW06] uses a model in which clocks drift, just like in this lecture. At least for this line of work, this goes back to a work by Fan and Lynch on *gradient clock synchronization,* [FL06] which shows that it is *not* possible to distribute the global skew of $\Omega(uD)$ "nicely" so that the skew between adjacent nodes is $\mathcal{O}(u)$ at all times; the possibility to "introduce skew on the fly" is essential for this observation. More on this in the next two lectures!

Bibliography

[BW01] Saâd Biaz and Jennifer Lundelius Welch. Closed Form Bounds for Clock Synchronization under Simple Uncertainty Assumptions. *Information Processing Letters*, 80:151–157, 2001.

[FL06] Rui Fan and Nancy Lynch. Gradient Clock Synchronization. *Distributed Computing*, 18(4):255–266, 2006.

[LL84] Jennifer Lundelius and Nancy Lynch. An Upper and Lower Bound for Clock Synchronization. *Information and Control*, 1984.

[LW06] Thomas Locher and Roger Wattenhofer. Oblivious Gradient Clock Synchronization. In *Proc. 20th Symposium on Distributed Computing (DISC)*, pages 520–533, 2006.

Lecture 2

Gradient Clock Synchronization

In the previous lesson, we proved essentially matching upper and lower bounds on the worst-case global skew for the clock synchronization problem. We saw that during an execution of the Max algorithm (Algorithm 1.2), all logical clocks in all executions eventually agree up to an additive term of $\mathcal{O}(uD)$ (ignoring other parameters). The lower bound we proved in Section 1.3 shows that global skew of $\Omega(uD)$ is unavoidable for any algorithm in which clocks run at an amortized constant rate, at least in the worst case. In our lower bound construction, the two nodes v and w that achieved the maximal skew were distance D apart. However, the lower bound did not preclude neighboring nodes from remaining closely synchronized throughout an execution. In fact, this is straightforward if one is willing to slow down clocks arbitrarily (or simply stop them), even if the amortized rate is constant.

Today, we look into what happens if one requires that clocks progress at a constant rate at all times. In many applications, it is sufficient that neighboring clocks are closely synchronized, while nodes that are further apart are only weakly synchronized. To model this situation, we introduce the *gradient clock synchronization (GCS) problem.* Intuitively, this means that we want to ensure a small skew between neighbors despite maintaining “proper” clocks. That is, we minimize the *local skew* under the requirement that logical clocks always run at least at rate 1.

2.1 Formalizing the Problem

Let $G = (V, E)$ be a network. As in the previous lecture, each node $v \in V$ has a hardware clock $H_v : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ that satisfies for all $t, t' \in \mathbb{R}_0^+$ with $t' < t$

$$t - t' \leq H_v(t) - H_v(t') \leq \vartheta(t - t') \ .$$

Again, we denote by $h_v(t)$ the rate of $H_v(t)$ at time t, i.e., $1 \leq h(t) \leq \vartheta$ for all $t \in \mathbb{R}_0^+$. Recall that each node v computes a logical clock $L_v \colon \mathbb{R}_0^+ \to \mathbb{R}_0^+$ from its hardware clock and messages received from neighbors. During an execution $\mathcal{E}$, for each edge $e = \{v, w\} \in E$, we define the *local skew* of e at time t to be

$\mathcal{L}_e(t) = |L_v(t) - L_w(t)|$. The *gradient skew at time* t in the network, denoted $\mathcal{L}(t)$, is the largest local skew across any edge: $\mathcal{L}(t) = \max_{e \in E} \mathcal{L}_e(t)$. Finally, the *gradient skew over an execution* $\mathcal{E}$ is defined to be

$$\mathcal{L} = \sup_{t \in \mathbb{R}_0^+} \{\mathcal{L}(t)\} \; .$$

The goal of the *gradient clock synchronization problem* is to minimize $\mathcal{L}$ for any possible execution $\mathcal{E}$.

Attention: In order to simplify our presentation of the gradient clock synchronization problem, we abstract away from the individual messages and message delays from the previous chapter. Instead, we assume that throughout an execution, each node v maintains an estimate of its neighbors' logical clocks. Specifically, for each neighbor $w \in N_v$, v maintains a variable $\tilde{L}_w^v(t)$. The parameter δ represents the *error* in the estimates: for all $\{v, w\} \in E$ and $t \in \mathbb{R}_0^+$, we have

$$L_w(t) \geq \tilde{L}_w^v(t) > L_w(t) - \delta \; . \tag{2.1}$$

When the node v is clear from context, we will omit the superscript v, and simply write $\tilde{L}_w$.

In order to obtain the estimates $\tilde{L}_w^v(t)$, each node w periodically broadcasts its logical clock value to its neighbors. Each neighbor v then computes $\tilde{L}_w^v(t)$ using the known bounds on message delays, and increases $\tilde{L}_w^v$ at rate h_v/ϑ between messages from w. Thus, an upper bound on the error parameter δ can be computed as a function of u (the uncertainty in message delay), ϑ (the maximum clock drift), T (the frequency of broadcasts), and μ (a parameter determining how fast logical clocks may run, see below); you do this in the exercises.

To focus on the key ideas, we make another simplifying abstraction: Instead of analyzing the global skew, we assume that it is taken care of and plug in $\mathcal{G}$ as a parametrized upper bound. You will address this issue as an exercise, too.

2.2 Averaging Protocols

In this section, we consider a natural strategy for achieving gradient clock synchronization: trying to bring the own logical clock to the average value between the neighbors whose clocks are furthest ahead and behind, respectively. Specifically, each node can be in either *fast mode* or *slow mode*. If a node v detects that its clock is behind the average of its neighbors, it will run in fast mode, and increase its logical clock at a rate faster than its hardware clock by a factor of $1 + \mu$, where μ is some appropriately chosen constant. On the other hand, if v's clock is at least the average of its neighbors, it will run in slow mode, increasing its logical clock only as quickly as its hardware clock. Note that this strategy results in logical clocks that behave like "real" clocks of drift $\vartheta' = \vartheta(1 + \mu) - 1$. If $\mu \in \mathcal{O}(\vartheta)$, these clocks are roughly as good as the original hardware clocks.

The idea of switching between fast and slow modes gives a well-defined protocol if neighboring clock values are known precisely,[1] however ambiguity

[1] There is one issue of pathological behavior in which nodes could switch infinitely quickly between fast and slow modes. This can be avoided by introducing a small threshold δ so that a node only changes, say, from slow to fast mode if it detects that its clock is δ time units behind the average.

arises in the presence of uncertainty.

We consider two natural ways of dealing with the uncertainty. Set $L_{N_v}^{\max}(t) := \max_{w \in N_v}\{L_w\}$ and $L_{N_v}^{\min}(t) := \min_{w \in N_v}\{L_w\}$.

Aggresive strategy: each v computes an *upper bound* on the average between $L_{N_v}^{\max}$ and $L_{N_v}^{\min}$, and determines whether to run in fast or slow mode based on this upper bound;

Conservative strategy: each v computes a *lower bound* on the average between $L_{N_v}^{\max}$ and $L_{N_v}^{\min}$ and determines the mode accordingly.

We will see that, in fact, both strategies yield terrible results, but for opposite reasons. In Section 2.3, we will derive an algorithm that strikes an appropriate balance between both stragies, with impressive results!

Aggressive Averaging

Here we analyze the aggressive averaging protocol described above. Specifically, each node $v \in V$ computes an upper bound on the average of its neighbors' logical clock values:

$$\tilde{L}_v^{\mathrm{up}}(t) = \frac{\max_{w \in N_v}\{\tilde{L}_w\} + \min_{w \in N_v}\{\tilde{L}_w\}}{2} + \delta \geq \frac{L_{N_v}^{\max} + L_{N_v}^{\min}}{2} .$$

The algorithm then increases the logical clock of v at a rate of $h_v(t)$ if $L_t(t) > \tilde{L}_v^{\mathrm{up}}(t)$, and a rate of $(1+\mu)h_v(t)$ otherwise. We show that the algorithm performs poorly for any choice of $\mu \geq 0$.

Claim 2.1. *Consider the aggressive averaging protocol on a path network of diameter D, i.e., $V = \{v_i \,|\, i \in [D+1]\}$ and $E = \{v_i, v_{i+1}\} \,|\, i \in [D]$. Then there exists an execution $\mathcal{E}$ such that the gradient skew satisfies $\mathcal{L} \in \Omega(\delta D)$.*

Proof Sketch. Throughout the execution, we will assume that all clock estimates are correct: for all $v \in V$ and $w \in N_v$, we have $\tilde{L}_v^w(t) = L_w(t)$. This means for all $i \in [D] \setminus \{0\}$ that $\tilde{L}_{v_i}^{\mathrm{up}}(t) = (L_{v_{i-1}}(t) + L_{v_{i+1}}(t))/2 + \delta$, whereas $\tilde{L}_{v_0}^{\mathrm{up}}(t) = L_{v_1}(t) + \delta$ and $\tilde{L}_{v_D}^{\mathrm{up}} = L_{v_{D-1}}(t) + \delta$. Initially, the hardware clock rate of node v_i is $1 + \frac{i(\vartheta-1)}{D}$. Thus, even though all nodes immediately "see" that skew is building up, they all set their clock rates to fast mode in order to catch up in case they underestimate their neighbors' clock values.

Now let's see what happens to the logical clocks in this execution. While nodes are running fast, skew keeps building up, but the property that $L_{v_i}(t) = (L_{v_{i+1}}(t) - L_{v_{i-1}}(t))$ is maintained at nodes $i \in [D] \setminus \{0\}$. In this state, v_0 — despite running fast — has no way of catching up to v_1. However, at time $\tau_0 := \frac{\delta D}{(1+\mu)(\vartheta-1)}$ we would have that $L_{v_D}(\tau_0) = L_{v_{D-1}}(\tau_0) + \delta = \tilde{L}_{v_D}^{\mathrm{up}}(\tau_0)$ and v_D would stop running fast. We set $t_0 := \tau_0 - \varepsilon$ for some arbitrarily small $\varepsilon > 0$ and set $h_{v_D}(t) := h_{v_{D-1}}(t)$ for all $t \geq t_0$. Thus, all nodes would remain in fast mode until the time $\tau_1 := t_0 + \frac{\delta D}{(1+\mu)(\vartheta-1)}$ when we had $L_{v_{D-1}}(\tau_1) = \tilde{L}_{v_{D-1}}^{\mathrm{up}}(\tau_1)$. We set $t_1 := \tau_1 - \varepsilon$ and proceed with this construction inductively. Note that, with every hop, the local skew increases by (almost) 2δ, as this is the additional skew that L_{v_i} must build up to $L_{v_{i-1}}$ when $L_{v_{i+1}} = L_{v_i}$ in order to increase $\tilde{L}_{v_i}^{\mathrm{up}} - L_{v_i}$ by δ, i.e., for v_i to stop running fast. As ε is arbitrarily small, we build up a local skew that is arbitrarily close to $(2D-1)\delta$. □

Remarks:

- The algorithm is also bad in that the above execution results in a global skew of $\Omega(\delta D^2)$.
- This could be fixed fairly easily, but without further changes still a large local skew could build up.
- The above argument can be generalized to arbitrary graphs, by taking two nodes $v, w \in V$ in distance D and using the function $d(x) = d(x, v) - d(x, w)$, just as in Lemma 1.5.

Conservative Averaging

Let's be more careful. Now each node $v \in V$ computes a *lower* bound on the average of its neighbors' logical clock values:

$$\tilde{L}_v^{\mathrm{up}}(t) = \frac{\max_{w \in N_v}\{\tilde{L}_w\} + \min_{w \in N_v}\{\tilde{L}_w\}}{2} \leq \frac{L_{N_v}^{\max} + L_{N_v}^{\min}}{2} .$$

The algorithm then increases the logical clock of v at a rate of $h_v(t)$ if $L_v(t) > \tilde{L}_v^{\mathrm{up}}(t)$, and a rate of $(1+\mu)h_v(t)$ otherwise. Again, the algorithm fails to achieve a small local skew.

Claim 2.2. *Consider the conservative averaging protocol on a path network of diameter D. Then there exists an execution $\mathcal{E}$ such that the gradient skew satisfies $\mathcal{L} \in \Omega(\delta D)$.*

Proof Sketch. We do the same as for the aggressive strategy, except that now for each $v \in V$, $w \in N_w$, and time t, we rule that $\tilde{L}_w(t) = L_w(t) - \delta + \varepsilon$ for some arbitrarily small $\varepsilon > 0$. Thus, all nodes are initially in slow mode. We inductively change hardware clock speeds just before nodes would switch to fast mode, building up the exact same skews between logical clocks as in the previous execution. The only difference is that now it does not depend on μ how long this takes! □

Remarks:

- It seems as if we just can't do things right. Both the aggressive and the conservative strategy do not result in a proper response to the gobal distribution of clock values.
- Surprisingly, mixing the two strategies works! We study this during the remainder of the lecture.

2.3 GCS Algorithm

The high-level strategy of the algorithm is as follows. As above, at each time each node can be either in *slow mode* or *fast mode.* In slow mode, a node v will increase its logical clock at rate $h_v(t)$. In fast mode, v will increase its logical clock at rate $(1 + \mu)h_v(t)$. The parameter μ will be chosen large enough for nodes whose logical clocks are behind to be able to catch up to other nodes. The conditions for a node to switch from slow to fast or vice versa are simple, but perhaps unintuititve. In what follows, we first describe "ideal" conditions to switch between modes. In the ideal behavior, each node knows exactly the logical clock values of its neighbors. Since the actual algorithm only has access to estimates of neighboring clocks, we then describe fast and slow triggers for switching between modes that can be implemented in our model for GCS. We conclude the section by proving that the triggers do indeed implement the conditions.

Fast and Slow Conditions

Definition 2.3 (**FC**: Fast Mode Condition)**.** *We say that a node $v \in V$ satisfies the* fast mode condition (**FC**) *at time $t \in \mathbb{R}_0^+$ if there exists $s \in \mathbb{N}$ such that:*

***FC** 1:* $\exists x \in N_v \colon L_x(t) - L_v(t) \geq 2s\delta\,;$

***FC** 2:* $\forall y \in N_v \colon L_v(t) - L_y(t) \leq 2s\delta\,.$

Informally, **FC** 1 says that v has a neighbor x whose logical clock is significantly ahead of $L_v(t)$, while **FC** 2 stipulates that none of v's neighbors' clocks is too far behind $L_v(t)$. In particular, if **FC** is satisfied with $x \in N_v$ satisfying **FC** 1, then the local skew across $\{v, x\}$ is at least $2s\delta$, where L_x is at least $2s\delta$ time units ahead of L_v. Since none of v's neighbors are running more than $2s\delta$ units behind L_v, v can decrease the maximum skew with its neighbors by increasing its logical clock.

The slow mode condition below is dual to **FC**. It essentially gives conditions under which v could decrease the maximum skew in its neighborhood by decreasing its logical clock.

Definition 2.4 (**SC**: Slow Mode Condition)**.** *We say that a node $v \in V$ satisfies the* slow mode condition *(or **SC**) at time $t \in \mathbb{R}_0^+$ if there exists $s \in \mathbb{N}$ such that:*

***SC** 1:* $\exists x \in N_v \colon L_v(t) - L_x(t) \geq (2s-1)\delta\,;$

***SC** 2:* $\forall y \in N_v \colon L_y(t) - L_v(t) \leq (2s-1)\delta\,.$

Substracting an additional δ in **SC** 1 and **SC** 2 ensures that conditions **FC** and **SC** are mutually exclusive. Together, the conditions mean that, if in doubt, the algorithm alternates between aggressively seeking to reduce skew towards neighbors that are ahead (**FC**) and conservatively avoiding to build up additional skew to neighbors that are behind (**SC**), depending on the currently observed average skew.

Fast and Slow Triggers

While the fast and slow mode conditions described in the previous section are well-defined (and mutually exclusive), uncertainty on neighbors' clock values prevents an algorithm from checking the conditions directly. Here we define corresponding *triggers* that our computational model does allow us to check.

The separation of δ between the conditions is just enough for this purpose. As we assumed that clock values are never overestimated, but may be underestimated by δ, the fast mode trigger needs to shift its thresholds by δ.

Definition 2.5 (**FT**: Fast Mode Trigger)**.** *We say that $v \in V$ satisfies the* fast mode trigger (**FT**) *at time $t \in \mathbb{R}_0^+$ if there exists an integer $s \in \mathbb{N}$ such that:*

***FT** 1: $\exists x \in N_v\colon \tilde{L}_x(t) - L_v(t) > (2s-1)\delta$;*

***FT** 2: $\forall y \in N_v\colon L_v(t) - \tilde{L}_y(t) < (2s+1)\delta$.*

Definition 2.6 (**ST**: Slow Mode Trigger)**.** *We say that a node $v \in V$ satisfies the* slow mode trigger *(or **ST**) at time $t \in \mathbb{R}_0^+$ if there exists $s \in \mathbb{N}$ such that:*

***ST** 1: $\exists x \in N_v\colon L_v(t) - \tilde{L}_x(t) \geq (2s-1)\delta$;*

***ST** 2: $\forall y \in N_v\colon \tilde{L}_y(t) - L_v(t) \leq (2s-1)\delta$.*

Before we formally describe the GCS algorithm, we give two preliminary results about the fast and slow mode triggers. The first result claims that **FT** and **ST** cannot simultaneously be satisfied by the same node. The second shows that **FT** and **ST** implement **FC** and **SC**, respectively. That is, if the fast (resp. slow) mode condition is satisfied, then the fast (resp. slow) mode trigger is also satisfied.

Lemma 2.7. *No node $v \in V$ can simultaneously satisfy **FT** and **ST**.*

Proof. Suppose v satisfies **FT**, i.e., there is $s \in \mathbb{N}$ so that there is some $x \in N_v$ such that $\tilde{L}_x(t) - L_v(t) > (2s-1)\delta$ and for all $y \in N_v$ we have $L_v(t) - \tilde{L}_y(t) < (2s+1)\delta$. Consider $s' \in \mathbb{N}$. If $s' > s$, then for all $y \in N_v$ we have that

$$L_v(t) - \tilde{L}_x(t) < (2s+1)\delta \leq (2s'-1)\delta\,,$$

so **ST** 1 is not satisfied for s'. If $s' \leq s$, then there is some $x \in N_v$ so that

$$\tilde{L}_x(t) - L_v(t) > (2s-1)\delta \geq (2s'-1)\delta\,,$$

so **ST** 2 is not satisfied for s'. Hence, **ST** is not satisfied. □

Lemma 2.8. *Suppose $v \in V$ satisfies **FC** (resp. **SC**) at time t. Then v satisfies **FT** (resp. **SC**) at time t.*

Proof. Suppose **FC** holds (at time t). Then, by (2.1), there is some $s \in \mathbb{N}$ such that

$$\exists x \in \mathbb{N}_v\colon \tilde{L}_x(t) - L_v(t) > L_x(t) - \delta - L_v(t) \geq (2s-1)\delta$$

and

$$\forall y \in \mathbb{N}_v\colon L_v(t) - \tilde{L}_y(t) < L_v(t) - L_y(t) + \delta \leq (2s+1)\delta\,,$$

i.e., **FT** holds. Similarly, if **SC** holds, (2.1) yields that

$$\exists x \in \mathbb{N}_v\colon L_v(t) - \tilde{L}_x(t) \geq L_v(t) - L_x(t) \geq (2s-1)\delta$$

and

$$\forall y \in \mathbb{N}_v\colon \tilde{L}_y(t) - L_x(t) \leq L_y(t) - L_v(t) \leq (2s-1)\delta$$

for some $s \in \mathbb{N}$, establishing **ST**. □

We now describe the GCS algorithm. Each node v initializes its logical clock to its hardware clock value. It continuously checks if the fast (resp. slow) mode trigger is satisfied. If so, it increases its logical clock at a rate of $(1+\mu)h_v(t)$ (resp. $h_v(t)$). Pseudocode is presented in Algorithm 2.1. The algorithm itself is simple, but the analysis of the algorithm (presented in the following section) is rather delicate.

Algorithm 2.1: GCS algorithm

```
$L_v(0) := H_v(0)$
$r := 1$
at all times $t$ do the following
if $FT$ then
    $r := 1 + \mu$                                  // $v$ is in fast mode
if $ST$ then
    $r := 1$                                        // $v$ is in slow mode
increase $L_v$ at rate $rh_v(t)$
```

Remarks:

- In fact, when neither **FT** nor **ST** hold, the logical clock may run at any speed from the range $[h_v(t), (1+\mu)h_v(t)]$.
- In order for the algorithm to be implementable, δ should leave some wiggle space. We expressed this by having (2.1) include a strict inequality, but if the inequality can become arbitrarily tight, the algorithm may have to switch between slow and fast mode arbitrarily fast.
- For technical reasons, we will assume that logical clocks are differentiable. Thus, $l_v := \frac{d}{dt}L_v$ exists and is between 1 and $\vartheta(1+\mu)$ at all times. It is possible to prove the guarantees of the algorithm without this assumption, but all this does is making the math harder.
- Even with this assumption, we still need Lemma A.1. This is not a mathematics lecture, but as we couldn't find any suitable reference, the lemma and a proof is given in the appendix.

2.4 Analysis of the GCS Algorithm

We now show that the GCS algorithm (Algorithm 2.1) indeed achieves a small local skew, which is expressed by the following theorem.

Theorem 2.9. *For every network G and every execution $\mathcal{E}$ in which $H_v(0) - H_w(0) \leq \delta$ for all edges $\{v, w\} \in E$, the GCS algorithm achieves a gradient skew of $\mathcal{L} \leq 2\delta\lceil\log_\sigma \mathcal{G}/\delta\rceil$, where $\sigma := \mu/(\vartheta - 1)$.*

In order to prove Theorem 2.9, we analyze the *average* skew over paths in G of various lengths. For long paths of $\Omega(D)$ hops, we will simply exploit that $\mathcal{G}$ bounds the skew between *any* pair of nodes. For successively shorter paths, we inductively show that the average skew between endpoints cannot increase too quickly: reducing the length of a path by factor σ can only increase the skew between endpoints by an additive constant term. Thus, paths of constant length (in particular edges) can only have a skew that is logarithmic in the network diameter.

Leading Nodes

We start by showing that skew cannot build up too quickly. This is captured by the following functions.

Definition 2.10 (Ψ and Leading Nodes)**.** *For each $v \in V$, $s \in \mathbb{N}$, and $t \in \mathbb{R}_0^+$, we define*

$$\Psi_v^s(t) = \max_{w \in V}\{L_w(t) - L_v(t) - (2s-1)\delta d(v,w)\}\,,$$

where $d(v,w)$ denotes the distance between v and w in G. Moreover, set

$$\Psi^s(t) = \max_{w \in V}\{\Psi_w^s(t)\}\,.$$

Finally, we say that $w \in V$ is a leading node *if there is some $v \in V$ so that*

$$\Psi_v^s(t) = L_w(t) - L_v(t) - (2s-1)\delta d(v,w) > 0\,.$$

We will show that $\Psi^s(t) \leq \mathcal{G}/\sigma^s$ for each $s \in \mathbb{N}$ and all times t. For $s = \lceil \log_\sigma \mathcal{G}/\delta \rceil$, this yields that

$$L_v(t) - L_w(t) - (2s-1)\delta \leq \mathcal{G}/\sigma^s \leq \delta \quad \Rightarrow \quad L_v(t) - L_w(t) \leq 2\delta \lceil \log_\sigma \mathcal{G}/\delta \rceil\,.$$

The definition of Ψ_v^s is closely related to the slow mode condition **SC**. It makes sure that leading nodes are always in slow mode.

Lemma 2.11 (Leading Lemma)**.** *Suppose $w \in V$ is a leading node at time t. Then w satisfies **SC** and **ST**.*

Proof. As w is a leading node at time t, there are $s \in \mathbb{N}$ and $v \in V$ so that

$$\Psi_v^s(t) = L_w(t) - L_v(t) - (2s-1)\delta d(v,w) > 0\,.$$

In particular, $L_w(t) > L_v(t)$, so $w \neq v$. For any $y \in V$, we have that

$$L_w(t) - L_v(t) - (2s-1)\delta d(v,w) = \Psi_v^s(t) \geq L_y(t) - L_v(t) - (2s-1)\delta d(y,w)\,.$$

Rearranging this yields

$$L_w(t) - L_y(t) \geq (2s-1)\delta(d(v,w) - d(y,w))\,.$$

In particular, for any $y \in N_v$, $d(v,w) \geq d(y,w) - 1$ and hence

$$L_y(t) - L_w(t) \leq (2s-1)\delta\,,$$

i.e., **SC** 2 holds at w. Now consider $x \in N_v$ so that $d(x,w) = d(v,w) - 1$; as $v \neq w$, such a node exists. We get that

$$L_w(t) - L_y(t) \geq (2s-1)\delta\,,$$

showing **SC** 1. By Lemma 2.8, w then also satisfies **ST** at time t. □

This can readily be translated into a bound on the growth of Ψ_w^s whenever it is positive.

Lemma 2.12 (Wait-up Lemma)**.** *Suppose $w \in V$ satisfies $\Psi_w^s(t) > 0$ for all $t \in (t_0, t_1]$. Then*

$$\Psi_w^s(t_1) \leq \Psi_w^s(t_0) - (L_w(t_1) - L_w(t_0)) + \vartheta(t_1 - t_0).$$

Proof. Fix $w \in V$, $s \in \mathbb{N}$ and $(t_0, t_1]$ as in the hypothesis of the lemma. For $v \in V$ and $t \in (t_0, t_1]$, define the function $f_v(t) = L_v(t) - (2s-1)\delta d(v,w)$. Observe that

$$\max_{v \in V}\{f_v(t)\} - L_w(t) = \Psi_w^s(t)\,.$$

Moreover, for any v satisfying $f_v(t) = L_w(t) + \Psi_w^s(t)$, we have that $L_v(t) - L_w(t) - (2s-1)\delta d(v,w) = \Psi_w^s(t) > 0$. Thus, Lemma 2.11 shows that v is in slow mode at time t. As (we assume that) logical clocks are differentiable, so is f_v, and it follows that $\frac{d}{dt}f_v(t) \leq \vartheta$ for any $v \in V$ and time $t \in (t_0, t_1]$ satisfying that $f_v(t) = \max_{x \in V}\{f_x(t)\}$. By Lemma A.1, it follows that $\max_{v \in V}\{f_v(t)\}$ grows at most at rate ϑ:

$$\max_{v \in V}\{f_v(t_1)\} \leq \max_{v \in V}\{f_v(t_0)\} + \vartheta(t_1 - t_0)\,.$$

We conclude that

$$\begin{aligned}\Psi_w^s(t_1) - \Psi_w^s(t_0) &= \max_{v \in V}\{f_v(t_1)\} - L_w(t_1) - (\max_{v \in V}\{f_v(t_0)\} - L_w(t_0))\\ &\leq -(L_w(t_1) - L_w(t_0)) + \vartheta(t_1 - t_0)\,,\end{aligned}$$

which can be rearranged into the claim of the lemma. □

Trailing Nodes

As $L_w(t_1) - L_w(t_0) \geq t_1 - t_0$ at all times, Lemma 2.15 shows that Ψ^s cannot grow faster than at rate $\vartheta - 1$ when it is positive. This buys us some time, but we need to show that w will make sufficient progress before Ψ^s grows larger than the desired bound. The approach to showing this is very similar to the one for Lemma 2.12, where now we need to exploit the fast mode condition **FC**.

Definition 2.13 (Trailing Nodes). *We say that $w \in V$ is a* trailing node *at time t, if there is some $s \in \mathbb{N}$ and a node v such that*

$$L_v(t) - L_w(t) - 2s\delta d(v,w) = \max_{x \in V}\{L_v(t) - L_x(t) - 2s\delta d(v,x)\} > 0\,.$$

Lemma 2.14 (Trailing Lemma). *Suppose $w \in V$ is a trailing node at time t. Then w satisfies **FC** and **FT**.*

Proof. Let s and v be such that

$$L_v(t) - L_w(t) - 2s\delta d(v,w) = \max_{x \in V}\{L_v(t) - L_x(t) - 2s\delta d(v,x)\} > 0\,.$$

In particular, $L_v(t) > L_w(t)$, implying that $v \neq w$. For $y \in V$, we have that

$$L_v(t) - L_w(t) - 2s\delta d(v,w) \geq L_v(t) - L_y(t) - 2s\delta d(v,y)$$

and thus for all neighbors $y \in N_w$ that

$$L_y(t) - L_w(t) + 2s\delta(d(v,y) - d(v,w)) \geq 0\,.$$

It follows that

$$\forall y \in N_v\colon L_w(t) - L_y(t) \leq 2s\delta\,,$$

i.e., **FC** 2 holds. As $v \neq w$, there is some node $x \in N_v$ with $d(v,x) = d(v,w) - 1$. We obtain that

$$\exists x \in N_v\colon L_y(t) - L_w(t) \geq 2s\delta\,,$$

showing **FC** 1. By Lemma 2.8, w thus also satisfies **FT** at time t. □

Using this, we can show that if $\Psi^s_w(t_0) > 0$, w will eventually catch up. How long this takes can be expressed in terms of $\Psi^{s-1}(t_0)$, or, if $s = 1$, $\mathcal{G}$.

Lemma 2.15 (Catch-up Lemma)**.** *Let $s \in \mathbb{N}$ and t_0, t_1 be times. If $s = 1$, suppose that $t_1 \geq t_0 + \mathcal{G}/\mu$; otherwise, suppose that $t_1 \geq t_0 + \Psi^{s-1}(t_0)/\mu$. Then, for any $w \in V$,*

$$L_w(t_1) - L_w(t_0) \geq t_1 - t_0 + \Psi^s_w(t_0)\,.$$

Proof. Choose $v \in V$ such that

$$\Psi^s_w(t_0) = L_v(t_0) - L_w(t_0) - (2s-1)\delta d(v,w) > 0\,.$$

Define $f_x(t) \coloneqq L_v(t_0) + (t - t_0) - L_x(t) - (2s-2)\delta d(v,x)$ for $x \in V$ and observe that $\Psi^s_w(t_0) \leq f_w(t_0)$. Hence, if $\max_{x\in V}\{f_x(t)\} \leq 0$ for some $t \in [t_0, t_1]$, then

$$\begin{aligned} L_w(t_1) - L_w(t) - (t_1 - t) \geq 0 \geq f_w(t)& \\ &= L_v(t_0) + (t - t_0) - L_w(t) - (2s-2)\delta d(v,x) \\ &= f_w(t_0) + (t - t_0) - (L_w(t) - L_w(t_0)) \\ &\geq \Psi^s_w(t_0) + (t - t_0) - (L_w(t) - L_w(t_0))\,, \end{aligned}$$

which can be rearranged into the claim of the lemma.

To show this, consider any time $t \in [t_0, t_1]$ when $\max_{x\in V}\{f_x(t)\} > 0$ and let $y \in V$ be any node such that $\max_{x\in V}\{f_x(t)\} = f_y(t)$. Then y is trailing, as

$$\begin{aligned} &\max_{x\in V}\{L_v(t) - L_x(t) - (2s-2)\delta d(v,x)\} \\ &= L_v(t) - L_v(t_0) - (t - t_0) + \max_{x\in V}\{f_x(t)\} \\ &= L_v(t) - L_v(t_0) - (t - t_0) + f_y(t) \\ &= L_v(t) - L_y(t) - (2s-2)\delta d(v,y) \end{aligned}$$

and

$$L_v(t) - L_v(t_0) - (t - t_0) + \max_{x\in V}\{f_x(t)\} > L_v(t) - L_v(t_0) - (t - t_0) \geq 0\,.$$

Thus, by Lemma 2.14 y is in fast mode. As logical clocks are (assumed to be) differentiable, we get that $\frac{d}{dt} f_y(t) = 1 - l_y(t) \leq -\mu$.

Now assume for contradiction that $\max_{x\in V}\{f_x(t)\} > 0$ for all $t \in [t_0, t_1]$. Then, applying Lemma A.1 again, we conclude that

$$\max_{x\in V}\{f_x(t_0)\} > -(\max_{x\in V}\{f_x(t_1)\} - \max_{x\in V}\{f_x(t_0)\}) \geq \mu(t_1 - t_0)\,.$$

If $s = 1$, $\mu(t_1 - t_0) \geq \mathcal{G}$, contradicting the fact that

$$f_x(t_0) = L_v(t_0) - L_x(t_0) \leq \mathcal{G}$$

for all $x \in V$. If $s > 1$, then $\mu(t_1 - t_0) \geq \Psi^{s-1}(t_0)$. However, we have that

$$f_x(t_0) \leq L_v(t_0) - L_x(t_0) - (2s-3)\delta d(v,x) \leq \Psi^{s-1}(t_0)$$

for all $x \in V$. As this is a contradiction as well, the claim of the lemma follows. □

Putting Things Together

Theorem 2.16. *Assume that $H_v(0) - H_w(0) \leq \delta$ for all $\{v,w\} \in E$. Then, for all $s \in \mathbb{N}$, Algorithm 2.1 guarantees $\Psi^s(t) \leq \mathcal{G}/\sigma^s$, where $\sigma = \mu/(1-\vartheta)$.*

Proof. Suppose for contradiction that the statement of the theorem is false. Let $s \in \mathbb{N}$ be minimal such that there is a time t_1 for which $\Psi^s(t_1) = \mathcal{G}/\sigma^s + \varepsilon$ for some $\varepsilon > 0$. Thus, there is some $w \in V$ such that

$$\Psi^s_w(t_1) = \Psi^s(t_1) = \frac{\mathcal{G}}{\sigma^s} + \varepsilon\,.$$

Set $t_0 := \max\{t - \mathcal{G}/(\mu\sigma^{s-1}), 0\}$. Consider the time $t' \in [t_0, t_1]$ that is minimal with the property that $\Psi^s_w(t) > 0$ for all $t \in (t', t_1]$ (by continuity of Ψ^s_w such a time exists). Thus, we can apply Lemma 2.12 to this interval, yielding that

$$\Psi^s_w(t_1) \leq \Psi^s_w(t') + \vartheta(t_1 - t') - (L_w(t_1) - L_w(t')) \leq \Psi^s_w(t') + (\vartheta - 1)(t_1 - t')\,.$$

$\Psi^s_w(t')$ cannot be 0, as otherwise

$$\Psi^s_w(t_1) \leq (\vartheta - 1)(t_1 - t') \leq \frac{(\vartheta-1)}{\mu} \cdot \frac{\mathcal{G}}{\sigma^{s-1}} = \frac{\mathcal{G}}{\sigma^s}\,,$$

contradicting $\Psi^s_w(t_1) = \mathcal{G}/\sigma^s + \varepsilon$.

On the other hand, if $\Psi^s_w(t') > 0$, we must have $t' = t_0$ from the definition of t', and $t_0 \neq 0$ because

$$\begin{aligned} &\max_{v,w \in V}\{L_v(0) - L_w(0) - (2s-1)\delta d(v,w)\} \\ =\; &\max_{v,w \in V}\{H_v(0) - H_w(0) - (2s-1)\delta d(v,w)\} \\ \leq\; &\max_{v,w \in V}\{H_v(0) - H_w(0) - \delta d(v,w)\} \leq 0\,, \end{aligned}$$

as $H_v(0) - H_w(0) \leq \delta$ for all neighbors v, w by assumption. Hence, $t' = t_0 = t_1 - \mathcal{G}/(\mu\sigma^{s-1})$. If $s > 1$, the minimality of s yields that $\Psi^s(t_0) \leq \mathcal{G}/\sigma^{s-1}$. We apply Lemma 2.15 to level s, node w, and time $t' = t_0$, yielding that

$$\Psi^s_w(t_1) \leq \Psi^s_w(t_0) + \vartheta(t_1 - t_0) - (L_w(t_1) - L_w(t_0)) \leq (\vartheta - 1)(t_1 - t_0) \leq \frac{\mathcal{G}}{\sigma^s}\,,$$

again contradicting $\Psi^s_w(t_1) = \mathcal{G}/\sigma^s + \varepsilon$. Reaching a contradiction in all cases, we conclude that the statement of the theorem must indeed hold. □

Our main result, Theorem 2.9, is now immediate.

Proof of Theorem 2.9. We apply Theorem 2.16 and consider $s := \lceil \log_\sigma(\mathcal{G}/\delta) \rceil$. For any $\{v,w\} \in E$ and any time t, we thus have that

$$L_v(t) - L_w(t) - (2s-1)\delta = L_v(t) - L_w(t) - (2s-1)\delta d(v,w) \leq \Psi^s(t) \leq \frac{\mathcal{G}}{\sigma^s} \leq \delta\,.$$

Rearranging this and exchanging the roles of v and w, we obtain

$$\mathcal{L}(t) = \max_{\{v,w\} \in E}\{|L_v(t) - L_w(t)|\} \leq 2s\delta = 2\delta \lceil \log_\sigma(\mathcal{G}/\delta) \rceil\,. \qquad \square$$

What to Take Home

- A very simple algorithm achieves a surprisingly good local skew, even if clocks must advance at all times.

- The base of the logarithm in the bound is typically large. A cheap quartz oscillator guarantees $\vartheta - 1 \leq 10^{-5}$, while typically $u/d \geq 10^{-2}$. With a base of roughly 10^3, the logarithmic term usually remains quite small.

- The algorithmic idea is surprisingly versatile. It works if δ is different for each link, and with some modifications (to algorithm and analysis), adversarial changes in the graph can be handled.

Bibliographic Notes

Gradient clock synchronization was introduced by Fan and Lynch [FL06], who show a lower bound of $\Omega(\log(uD)/\log\log(uD))$ on the local skew. Some researchers found this result rather counter-intuitive, and it triggered a line of research seeking to resolve the question what precisely can be achieved. The first non-trivial upper bound was provided by Locher and Wattenhofer [LW06]. Their *blocking algorithm* bounds the local skew by $\mathcal{O}(\sqrt{\delta D})$. The first logarithmic bound on the local skew was given in [LLW08] and soon after improved to the algorithm presented here [LLW10]. However, the elegant way of phrasing it in terms of the fast and slow modes and conditions is due to Kuhn and Oshman [KO09].

The algorithmic idea underlying the presented solution turns out to be surprisingly robust and versatile. Essentially the same algorithm works for different uncertainties on the edges [KO09]. With a suitable method of carefully incorporating newly appearing edges, it can handle dynamic graphs [KLLO10] (this problem is introduced in [KLO11]), in the sense that edges that were continuously present for sufficiently long satisfy the respective guarantee on the skew between their endpoints. Recently, the approach has been independently discovered (twice!) for solving load balancing tasks that arise in certain packet routing problems [DLNO17, PR17].

Bibliography

[DLNO17] Stefan Dobrev, Manuel Lafond, Lata Narayanan, and Jaroslav Opatrny. Optimal local buffer management for information gathering with adversarial traffic. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2017, Washington DC, USA, July 24-26, 2017*, pages 265–274, 2017.

[FL06] Rui Fan and Nancy Lynch. Gradient Clock Synchronization. *Distributed Computing*, 18(4):255–266, 2006.

[KLLO10] Fabian Kuhn, Christoph Lenzen, Thomas Locher, and Rotem Oshman. Optimal Gradient Clock Synchronization in Dynamic Networks. *CoRR*, abs/1005.2894, 2010.

[KLO11] Fabian Kuhn, Thomas Locher, and Rotem Oshman. Gradient Clock Synchronization in Dynamic Networks. *Theory Comput. Syst.*, 49(4):781–816, 2011.

[KO09] Fabian Kuhn and Rotem Oshman. Gradient Clock Synchronization Using Reference Broadcasts. In *Proc. 13th Conference on Principles of Distributed Systems (OPODIS)*, pages 204–218, 2009.

[LLW08] Christoph Lenzen, Thomas Locher, and Roger Wattenhofer. Clock Synchronization with Bounded Global and Local Skew. In *Proc. 49th Symposium on Foundations of Computer Science (FOCS)*, pages 509–518, 2008.

[LLW10] Christoph Lenzen, Thomas Locher, and Roger Wattenhofer. Tight Bounds for Clock Synchronization. *J. ACM*, 57(2):8:1–8:42, 2010.

[LW06] Thomas Locher and Roger Wattenhofer. Oblivious Gradient Clock Synchronization. In *Proc. 20th Symposium on Distributed Computing (DISC)*, pages 520–533, 2006.

[PR17] Boaz Patt-Shamir and Will Rosenbaum. The space requirement of local forwarding on acyclic networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 13–22, 2017.

Lecture 3

Lower Bound on the Local Skew

In Chapter 1, we proved tight upper and lower bounds of $\Theta(D)$ for the global skew of any clock synchronization algorithm. However, the algorithms achieving optimal global skew had the undesireable feature that the maximal global skew could be attained between any pair of nodes in the network—even adjacent nodes. In Chapter 2, we developed a more refined algorithm that further controlled the gradient skew—the maximum skew between any pair of *adjacent* nodes. Specifically, the gradient clock synchronization (GCS) algorithm of Chapter 2 achieved a local skew of $\mathcal{O}(\delta \log D)$.

In this chapter, we address the question of whether the $\mathcal{O}(\delta \log D)$ skew upper bound for GCS can be improved. Since gradient clock synchronization is a local property (in the sense that the definition of gradient skew only references logical clocks of neighboring nodes), one may expect that a distributed algorithm may be able to achieve $\mathcal{O}(\delta)$ local skew. However, we will show that this is impossible: any GCS algorithm must incur local skew of $\Omega(u \log D)$ for some executions. Thus, the GSC algorithm of Chapter 2 is asymptotically optimal.

3.1 Lower Bound with Bounded Clock Rates

In this section, we first prove a lower bound assuming that each logical clock increases at a rate of at most $(1+\mu)h_v > 1$. That is, for all $v \in V$ and $t, t' \in \mathbb{R}_0^+$ with $t < t'$, we assume $L_v(t') - L_v(t) \leq (1+\mu)(H_v(t') - H_v(t))$.[1] We use the model of Chapter 1. Moreover, all logical clocks have a minimum rate of 1: for all $v \in V$ and $t, t' \in \mathbb{R}_0^+$ with $t < t'$, we have $L_v(t') - L_v(t) \geq t' - t$. Under these assumptions, we will prove the following theorem.

Theorem 3.1. *Any algorithm for the gradient clock synchronization problem with logical clock rates between* 1 *and* $(1+\mu)h_v$ *incurs a worst-case gradient skew of* $\mathcal{L} \geq (u/4 - (\vartheta - 1)d) \log_{\lceil \sigma \rceil} D$*, where* $\sigma := \mu/(\vartheta - 1)$.

[1] Note that this assumption does not allow for algorithms that increase their clocks discontinuously. For example, the argument does not apply to the max algorithm presented in Chapter 1.

To gain some intuition, assume that $(\vartheta-1)d \ll u$, so we can neglect this term. In order to prove Theorem 3.1, we first show that the adversary can build up a *hardware* clock skew of $\Omega(uk)$ between any pair of nodes in distance k in $\mathcal{O}(uk/(\vartheta-1))$ time, in an indistinguishable way. Specifically, for v and w in distance k, we get that $H_v^{(\mathcal{E}_v)}(t)-H_v^{(\mathcal{E}_1)}(t)\in\Omega(uk)$ for some time t, while $H_w^{(\mathcal{E}_v)}(t)=H_w^{(\mathcal{E}_1)}(t)$. By the minimum progress condition, this implies that the logical clock of v differs by at least $\Omega(uk)$ between the two executions. This is, in fact, a straightforward generalization of Lemma 1.5. The key difference is that we sacrifice a factor of 2 in the amount of skew we sneak in, so we can choose the pair of nodes between which we build the skew *after* examining what happens in $\mathcal{E}_1$, i.e., $\mathcal{E}_1$ provides no information regarding where the skew will "appear."

We can use this inductively as follows. Assuming that we know how to build up a skew of αuk between nodes in distance k (initially, $k\approx D$ and $\alpha=0$), we run a given GCS algorithm for $\mathcal{O}(uk'/(\vartheta-1))$ time with all hardware clock rates being 1 (that's the case in $\mathcal{E}_1$), where $k'\in\Theta(k/\sigma)$ (with constants chosen suitably). As logical clock rates are between 1 and $1+\mu$ in $\mathcal{E}_1$, the skew between the original nodes is still $\alpha uk-\mathcal{O}(u\mu k'/(\vartheta-1))=(\alpha-\mathcal{O}(1))uk$. Thus, there must be two nodes in distance k' with skew at least $(\alpha-\mathcal{O}(1))uk'$. If v is the node with the larger clock value, we now consider $\mathcal{E}_v$, in which the skew is by $\Omega(uk')$ larger. For the right choice of k', we end up with a path of length k' that has skew $(\alpha+\Omega(1))uk'$! We can repeat this up to $\Theta(\log_\sigma D))$ many times, yielding the desired lower bound.

Lemma 3.2. *Assume that $(\vartheta-1)d<u/2$ and set $t_0:=d(v,w)(u/(2(\vartheta-1))-d)$. For any algorithm, there is an execution $\mathcal{E}_1$ such that for any $v,w\in V$, there is an indistinguishable execution $\mathcal{E}_v$ satisfying that*

- *$H_x^{(\mathcal{E}_1)}(t)=t$ for all $x\in V$ and t,*
- *$H_v^{(\mathcal{E}_v)}(t)=H_v^{(\mathcal{E}_1)}(t)+d(v,w)(u/2-(\vartheta-1)d)$ for all $t\geq t_0$, and*
- *$H_w^{(\mathcal{E}_v)}(t)=t$ for all t.*

Proof. The proof is very similar to the one of Lemma 1.5. In both executions and for all $x\in V$, we set $H_x(0):=0$. Execution $\mathcal{E}_1$ is given by running the algorithm with all hardware clock rates being 1 at all times and the message delay from x to y being $d-u/2$.

Set

$$d(x):=\begin{cases}-d(v,w) & \text{if } d(x,w)-d(x,v)<-d(v,w)\\ d(v,w) & \text{if } d(x,w)-d(x,v)>d(v,w)\\ d(x,w)-d(x,v) & \text{else.}\end{cases}$$

Note that $|d(x)-d(y)|\leq 2$ for any $\{x,y\}\in E$. Moreover, $d(v)=d(v,w)$ and $d(w)=-d(v,w)$. In $\mathcal{E}_v$, we set the hardware clock rate of node $x\in V$ to $1+(\vartheta-1)(d(x)+d(v,w))/(2d(v,w))$ at all times $t\leq t_0$ and to 1 at all times $t>t_0$. This implies that

$$H_v^{(\mathcal{E}_v)}(t_0)=\vartheta t_0=H_v^{(\mathcal{E}_1)}(t_0)+d(v,w)\left(\frac{u}{2}-(\vartheta-1)d\right)\quad\text{and}$$
$$H_w^{(\mathcal{E}_v)}(t_0)=t_0\,.$$

As clock rates are 1 from time t_0 on, this means that the hardware clocks satisfy all stated constraints.

It remains to specify message delays and show that the two executions are indistinguishable. We achieve this by simply ruling that a message sent from some $x \in V$ to a neighbor $y \in N_x$ in $\mathcal{E}_v$ arrives at the same local time at y as it does in $\mathcal{E}_1$. By induction over the arrival sending times of messages, then indeed all nodes also send identical messages at identical local times in both executions, i.e., the executions are indistinguishable. However, it remains to prove that this results in all message delays being in the range $(d-u,d)$.

To see this, recall that for any $\{x,y\} \in E$, we have that $|d(x)-d(y)| \leq 2$. As clock rates are 1 after time t_0 and constant before, and all hardware clocks are 0 at time 0, the maximum difference between any two local times between neighbors is attained at time t_0. We compute

$$H_x^{(\mathcal{E}_v)}(t_0) - H_y^{(\mathcal{E}_v)}(t_0) = \frac{d(y)-d(x)}{2d(v,w)} \cdot (\vartheta-1)t_0 = \frac{d(y)-d(x)}{2}\left(\frac{u}{2}-(\vartheta-1)d\right).$$

In execution $\mathcal{E}_1$, a message sent from x to y at local time $H_x^{(\mathcal{E}_1)}(t) = t$ is received at local time $H_y^{(\mathcal{E}_1)}(t) = H_x^{(\mathcal{E}_1)}(t) + d - u/2$. If a message is sent at time t in $\mathcal{E}_v$, we have that

$$\begin{aligned}
H_y^{(\mathcal{E}_v)}(t+d) &\geq H_y^{(\mathcal{E}_v)}(t) + d \\
&= H_x^{(\mathcal{E}_v)}(t) + d + \frac{d(x)-d(y)}{2}\left(\frac{u}{2}-(\vartheta-1)d\right) \\
&> H_x^{(\mathcal{E}_v)}(t) + d - \frac{u}{2}
\end{aligned}$$

where the last inequality uses that $d(x)-d(y) \geq -2$ and that $u/2 > (\vartheta-1)d$ by assumption. On the other hand,

$$\begin{aligned}
H_y^{(\mathcal{E}_v)}(t+d-u) &< H_y^{(\mathcal{E}_v)}(t) + \vartheta d - u \\
&= H_x^{(\mathcal{E}_v)}(t) + \vartheta d - u + \frac{d(x)-d(y)}{2}\left(\frac{u}{2}-(\vartheta-1)d\right) \\
&\leq H_x^{(\mathcal{E}_v)}(t) + d - \frac{u}{2},
\end{aligned}$$

where the final inequality holds with equality if $d(x)-d(y)=2$ and thus also for $d(x)-d(y)<2$, as $u/2 > (\vartheta-1)d$. □

Proof of Theorem 3.1. Note that the claim is vacuous if $(\vartheta-1)d \geq u/4$, so we can assume the opposite in the following. Set $b := \lceil 2\sigma \rceil$ and $i_{\max} := \lfloor \log_b D \rfloor$. By induction over $i \in [i_{\max}+1]$, we show that we can build up a skew of $(i+2)(u/4-(\vartheta-1)d)d(v,w)$ between nodes $v,w \in V$ in distance $d(v,w) = b^{i_{\max}-i}$ at a time t_i in execution $\mathcal{E}^{(i)}$, such that after time t_i all hardware clock rates are 1 and all sent messages have delays of $d-u/2$.

We anchor the induction at $i=0$ by applying Lemma 3.2, choosing t_0 as in the lemma. We pick two nodes $v,w \in V$ in distance $b^{i_{\max}} \leq D$ of each other such that $L_v^{(\mathcal{E}_1)}(t_0) \geq L_w^{(\mathcal{E}_1)}(t_0)$. Now consider $\mathcal{E}_v$ for this choice of $v,w \in V$, which satisfies that $H_v^{(\mathcal{E}_v)}(t_0) = H_v^{(\mathcal{E}_1)}(t_0) + (u/2-(\vartheta-1)d)d(v,w)$ and $H_w^{(\mathcal{E}_v)}(t_0) = H_w^{(\mathcal{E}_1)}(t_0)$. By indistinguishability of the two executions and the minimum logical clock rate of 1, we get that

$$\begin{aligned} L_v^{(\mathcal{E}_v)}(t_0) - L_w^{(\mathcal{E}_v)}(t_0) &= L_v^{(\mathcal{E}_1)}\left(t_0 + \left(\frac{u}{2} - (\vartheta - 1)d\right) d(v,w)\right) - L_w^{(\mathcal{E}_1)}(t_0) \\ &\geq L_v^{(\mathcal{E}_1)}(t_0) + \left(\frac{u}{2} - (\vartheta - 1)d\right) d(v,w) - L_w^{(\mathcal{E}_1)}(t_0) \\ &\geq \left(\frac{u}{2} - (\vartheta - 1)d\right) d(v,w)\,. \end{aligned}$$

We obtain $\mathcal{E}^{(0)}$ by changing all hardware clock rates in $\mathcal{E}_v$ to 1 at time t_0 and all message delays of messages sent at or after time t_0 to $d - u/2$. As this does not affect the logical clock values at time t_0 — $\mathcal{E}^{(0)}$ is indistinguishable from $\mathcal{E}_v$ at $x \in V$ until local time $H_x^{(\mathcal{E}^{(0)})}(t_0)$ — this shows the claim for $i = 0$.

For the induction step from i to $i+1$, let $v, w \in V$, $\mathcal{E}^{(i)}$, and t_i be given by the induction hypothesis, i.e.,

$$L_v^{(\mathcal{E}^{(i)})}(t_i) - L_w^{(\mathcal{E}^{(i)})}(t_i) \geq (i+2)\left(\frac{u}{4} - (\vartheta - 1)d\right) d(v,w)\,,$$

and from time t_i on all hardware clock rates are 1 and sent messages have delay $d - u/2$. Note that the latter conditions mean that $\mathcal{E}^{(i)}$ behaves exactly like $\mathcal{E}_1$ from Lemma 3.2 from time t_i on, except that some messages sent at times $t < t_i$ may arrive during $[t_i, t_i + d)$. Hence, if we apply the same modifications to $\mathcal{E}^{(i)}$ as to $\mathcal{E}_1$, but starting from time $t_i + d$ instead of time 0, we can, for any $v', w' \in V$, construct an execution $\mathcal{E}_{v'}$ indistinguishable from $\mathcal{E}^{(i)}$, where

- $H_x^{(\mathcal{E}^{(i)})}(t) = H_x^{(\mathcal{E}^{(i)})}(t_i) + t - t_i$ for all $x \in V$ and $t \geq t_i$,
- $H_{v'}^{(\mathcal{E}_{v'})}(t) = H_{v'}^{(\mathcal{E}^{(i)})}(t) + d(v', w')(u/2 - (\vartheta - 1)d)$ for all times $t \geq t_i + d + (u/(2(\vartheta - 1)) - d)d(v', w')$, and
- $H_{w'}^{(\mathcal{E}_{v'})}(t) = H_{w'}^{(\mathcal{E}^{(i)})}(t_i) + t - t_i$ for all $t \geq t_i$.

Consider the logical clock values of v and w in $\mathcal{E}^{(i)}$ at time

$$t_{i+1} \coloneqq t_i + d + \left(\frac{u}{2(\vartheta - 1)} - d\right) \frac{d(v,w)}{b}\,.$$

Recall that $l_v(t) \geq h_v(t) \geq 1$ and $l_w(t) \leq (1+\mu) h_w(t)$ at all times t. As $h_w^{(\mathcal{E}^{(i)})}(t) = 1$ at times $t \geq t_i$, we get that

$$L_v^{(\mathcal{E}^{(i)})}(t_{i+1}) - L_w^{(\mathcal{E}^{(i)})}(t_{i+1}) \geq L_v^{(\mathcal{E}^{(i)})}(t_i) - L_w^{(\mathcal{E}^{(i)})}(t_i) - \mu(t_{i+1} - t_i)\,. \tag{3.1}$$

Recall that $d(v,w) = b^{i_{\max} - i}$ and that $b = \lceil 2\sigma \rceil$. We split up a shortest path from v to w in b subpaths of length $b^{i_{\max} - (i+1)}$. By the pidgeon hole principle, at least one of these paths must exhibit at least a $1/b$ fraction of the skew between v and w, i.e., there are $v', w' \in V$ with $d(v', w') = b^{i_{\max} - (i+1)} = d(v,w)/b$ so that

$$\begin{aligned}
&L_{v'}^{(\mathcal{E}^{(i)})}(t_{i+1}) - L_{w'}^{(\mathcal{E}^{(i)})}(t_{i+1})\\
\geq\ & \frac{L_{v}^{(\mathcal{E}^{(i)})}(t_{i+1}) - L_{w}^{(\mathcal{E}^{(i)})}(t_{i+1})}{b} \text{ by (3.1) we have:}\\
\geq\ & \frac{L_{v}^{(\mathcal{E}^{(i)})}(t_{i}) - L_{w}^{(\mathcal{E}^{(i)})}(t_{i}) - \mu(t_{i+1}-t_i)}{b}\\
=\ & \frac{L_{v}^{(\mathcal{E}^{(i)})}(t_{i}) - L_{w}^{(\mathcal{E}^{(i)})}(t_{i}) - \mu(d+(u/(2(\vartheta-1))-d)d(v',w'))}{b}\\
\geq\ & \frac{L_{v}^{(\mathcal{E}^{(i)})}(t_{i}) - L_{w}^{(\mathcal{E}^{(i)})}(t_{i}) - \mu u d(v',w')/(2(\vartheta-1))}{b}\\
\geq\ & \frac{L_{v}^{(\mathcal{E}^{(i)})}(t_{i}) - L_{w}^{(\mathcal{E}^{(i)})}(t_{i})}{b} - \frac{\mu}{2\sigma(\vartheta-1)} \cdot \frac{u}{2} \cdot d(v',w')\\
=\ & \frac{L_{v}^{(\mathcal{E}^{(i)})}(t_{i}) - L_{w}^{(\mathcal{E}^{(i)})}(t_{i})}{b} - \frac{u}{4} \cdot d(v',w')\\
\geq\ & \frac{(i+2)(u/4-(\vartheta-1)d)d(v,w)}{b} - \frac{u}{4}\cdot d(v',w')\\
=\ & \left((i+2)\left(\frac{u}{4}-(\vartheta-1)d\right)-\frac{u}{4}\right)d(v',w')\,.
\end{aligned}$$

In other words, as the average skew on a shortest path from v to w did not decrease by more than $u/4$, there most be some subpath of length $d(v,w)/b$ with at least the same average skew. Now we sneak in additional skew by advancing the (hardware and thus also logical) clock of v' using the indistinguishable execution $\mathcal{E}_{v'}$:

$$\begin{aligned}
&L_{v'}^{(\mathcal{E}_v)}(t_{i+1}) - L_{w'}^{(\mathcal{E}_v)}(t_{i+1})\\
&= L_{v'}^{(\mathcal{E}^{(i)})}\left(t_{i+1}+\left(\frac{u}{2}-(\vartheta-1)d\right)d(v',w')\right) - L_{w'}^{(\mathcal{E}^{(i)})}(t_{i+1})\\
&\geq L_{v'}^{(\mathcal{E}^{(i)})}(t_{i+1})+\left(\frac{u}{2}-(\vartheta-1)d\right)d(v',w') - L_{w'}^{(\mathcal{E}^{(i)})}(t_{i+1})\\
&\geq (i+3)\left(\frac{u}{4}-(\vartheta-1)d\right)d(v',w')\,.
\end{aligned}$$

This completes the induction. Plugging in $i = i_{\max}$ and noting that $\log b = \log\lceil 2\sigma\rceil \leq 1+\log\lceil\sigma\rceil$, we get an execution in which two nodes at distance $b^0 = 1$ exhibit a skew of at least

$$\begin{aligned}
(i_{\max}+2)\left(\frac{u}{4}-(\vartheta-1)d\right) &\geq \left(\frac{u}{4}-(\vartheta-1)d\right)(1+\log_b D)\\
&\geq \left(\frac{u}{4}-(\vartheta-1)d\right)\log_{\lceil\sigma\rceil} D\,. \qquad \square
\end{aligned}$$

Remarks:

- It is somewhat "bad form" to adapt Lemma 3.2 on the fly, as we did in the proof. However, the alternative of carefully defining partial executions, how to stitch them together, and proving indistinguishability results in this setting would mean to crack a nut with a sledgehammer.

- By making the base of the logarithm larger (i.e., making paths shorter more quickly), we can reduce the "loss" of skew in each step. Thus, we get a skew of $(u/2 - (\vartheta - 1)d - \varepsilon)$ per iteration, at the cost of reducing the number of iterations by a factor of $\log \sigma/(\log \sigma - \log \varepsilon^{-1})$.

- We can gain another factor of two by introducing skew more carefully. If we constract $\mathcal{E}_1$ so that messages "in direction of w" have delay (roughly) $d - u$ and messages "in direction of v" have delay d, we can hide u skew per hop, just like in Lemma 1.5. We favored the simpler construction to avoid additional bookkeeping.

- Overall, if $(\vartheta - 1)d \ll u$, $\sigma \gg 1$, and $\log_\sigma D \gg 1$, we can show a lower bound of $(u - \varepsilon) \log_\sigma D$ for some small $\varepsilon > 0$.

- Assuming a similar bunch of reasonable things and that $T \in \mathcal{O}(d)$ (i.e., message frequency is not the bottleneck in determining estimates), the asymptotically optimal choice of μ we computed in the exercises yields a skew of roughly $2u \log_\sigma D$ for our GCS algorithm. Thus, this lower bound shows that the algorithm is optimal up to a factor of roughly 2, provided $\sigma \gg 1$ and $(\vartheta - 1)d \ll u$. Dropping that $\sigma \gg 1$, we still get optimality up to a constant factor.

- So what of the case that $(\vartheta - 1)d$ is comparable to u or even larger? Recall that we have shown how to generate a better "logical hardware clock" in this case by bouncing messages back and forth between nodes. Using this idea (with some modifications and the occasional atrocity), one *could*, up to an additive $\mathcal{O}((\vartheta - 1)d)$, eliminate the dependence of the upper bound on $(\vartheta - 1)d$.

- As for a lower bound construction we can always pretend that clock drifts are actually smaller, e.g., $\vartheta' := \min\{\vartheta, 1 + u/(4d)\}$, the lower bound is asymptotically optimal in all cases...

- ...except for unbounded clock rates, which we will deal with next.

3.2 Lower Bound with Arbitrary Clock Rates

It can be shown that clock rates $l_v(t) \in \omega(1)$ do not help. That is, if $(\vartheta - 1)d < u/4$, we have that $\mathcal{L} \in \Omega(u \log_{1/(\vartheta-1)} D)$. However, the only (currently known) proof for this is tedious, to the point where it conveys little insight regarding what's going on. Hence, we will settle for a (much) simpler argument by Fan and Lynch showing a slightly weaker lower bound, followed by some intution as to why the stronger result is true as well.

We need a technical lemma stating that, provided that we leave some slack in terms of clock drifts and message delays, we can introduce $\Omega(u)$ hardware clock skew between any pair of neighbors in an indistinguishable manner. As this follows from repetition of previous arguments, we skip the proof.

Lemma 3.3. *Let $\mathcal{E}$ be any execution in which clock rates are at most $1+(\vartheta-1)/2$ and message delays are in the range $(d-3u/4, d-u/4)$. Then, for any $\{v, w\} \in E$ and sufficiently large times t, there is an indistinguishable execution $\mathcal{E}_v$ such that $L_v^{(\mathcal{E}_v)}(t) = L_v^{(\mathcal{E})}(t + u/4)$ and $L_w^{(\mathcal{E}_v)}(t) = L_w^{(\mathcal{E})}(t)$.*

Proof Sketch. The general idea is to use the remaining slack of $u/2$ to hide the additional skew, and the slack in the clock rates to introduce it. We can do this as slowly as needed, just as in the proof of Lemma 1.5. Again, we can choose the clock rates according to the function $d(x)$ defined in Lemma 3.2; as v and w are neighbors here, it can only take on values of -1, 0, or 1. □

This is all we need to generalize our lower bound to arbitrarily large logical clock rates.

Theorem 3.4. *Assume that $\vartheta \leq 2$. Any algorithm for the gradient clock synchronization problem with logical clock rates of at least 1 incurs a worst-case gradient skew of*

$$\mathcal{L} \in \Omega\left(\left(\frac{u}{4} - (\vartheta - 1)d\right) \log_{(\log D)/(\vartheta-1)} D\right).$$

Proof. Set $u' := u/2$, $d' := d - u/4$, and $\vartheta' := 1 + (\vartheta - 1)/2$. We perform the exact same construction as in Theorem 3.1, with three modifications. First, u, d, and ϑ are replaced by u', d', and ϑ'. Second, before starting the construction, we wait for sufficiently long so that Lemma 3.3 is applicable to all times when we actually "work," i.e., we let the algorithm run for the required time with hardware clock rates of 1 and message delays of $d' - u'/2$. Third, we assume that $\mu = \log_{1/(\vartheta-1)} D$ in the construction; if ever we attempt to use this (assumed) bound on the clock rates in an inequality and it does not hold, the construction fails.

Now two things can happen. The first is that the construction succeeds. Note that we may assume that $u'/4 > (\vartheta' - 1)d'$, as otherwise $u/4 < (\vartheta - 1)d$, i.e., nothing is to show. Thus, the construction shows a lower bound of

$$\begin{aligned}\left(\frac{u'}{4} - (\vartheta' - 1)d'\right) \log_{\lceil\sigma\rceil} D &> \left(\frac{u}{8} - \frac{(\vartheta - 1)d}{2}\right) \log_{\lceil\mu/(\vartheta'-1)\rceil} D \\ &\in \Omega\left(\left(\frac{u}{4} - (\vartheta - 1)d\right) \log_{\mu/(\vartheta-1)} D\right).\end{aligned}$$

As

$$\begin{aligned}\log_{\mu/(\vartheta-1)} D &= \frac{\log D}{\log \mu - \log(\vartheta - 1)} \\ &= \frac{\log D}{\log(\log D - \log(\vartheta - 1)) - \log(\vartheta - 1)} \\ &\in \Omega\left(\frac{\log D}{\log\log D - \log(\vartheta - 1}\right) \\ &= \Omega\left(\log_{(\log D)/(\vartheta-1)} D\right),\end{aligned}$$

the claim follows in this case.

On the other hand, if the construction fails, there is an index $i < i_{\max}$ for which (3.1) does not hold — this is the only place where we make use of the fact that logical clocks do not run faster than rate μ. Thus,

$$L_w^{(\mathcal{E}^{(i)})}(t_{i+1}) - L_w^{(\mathcal{E}^{(i)})}(t_i) > \mu(t_{i+1} - t_i)$$

for some $i < i_{\max}$. Recall that in the construction, $d(v,w) = b^{i_{\max}-i} \geq b$ and

$$t_{i+1} - t_i = d + \left(\frac{u}{2(\vartheta-1)} - d\right)\frac{d(v,w)}{b} > \frac{u}{2(\vartheta-1)} - d > \frac{u}{4(\vartheta-1)} \geq \frac{u}{4}\,.$$

Hence, there must be a time $t \geq t_i$ so that

$$L_w^{(\mathcal{E}^{(i)})}\left(t + \frac{u}{4}\right) - L_w^{(\mathcal{E}^{(i)})}(t) > \frac{\mu u}{4}\,.$$

Let $x \in N_w$ be arbitrary. By Lemma 3.3, we can construct an execution $\mathcal{E}_w$ so that

$$L_w^{(\mathcal{E}_w)}(t) = L_w^{(\mathcal{E}^{(i)})}\left(t + \frac{u}{4}\right) > L_w^{(\mathcal{E}^{(i)})}(t) + \frac{\mu u}{4}$$

and $L_x^{(\mathcal{E}_w)}(t) = L_x^{(\mathcal{E}^{(i)})}(t)$. Thus, in at least one of the executions, the local skew exceeds

$$\frac{\mu u}{8} = \frac{u}{8}\log_{1/(\vartheta-1)} D\,. \qquad \square$$

We conclude this chapter with the promised intuition regarding the influence of D on the base of the logarithm. Consider a path of length k with a skew of exactly α per hop, for a total of αk between its endpoints. Now suppose that an algorithm cleverly uses a large logical clock rate, perfectly reducing the skew at the same rate between any pair of neighbors. Consider the point in time when the skew has been reduced to, say, $\alpha - u/8$ per hop. The node in the middle of the path has increased its logical clock at half the rate of the endpoint that's catching up — and the nodes in between have been even faster! Denoting this rate by r, slipping in hardware clock skew at rate $\vartheta - 1$ means adding logical clock skew at rate at least $r(\vartheta-1)/2$. So, even if it takes factor r less time to reduce the skew to, say $\alpha - u/8$ per hop than it would for $\mu = 1$, it also takes factor $r/2$ less time to build up additional skew. We would end up with the same result!

Remarks:

- Unfortunately, molding this idea into a proof is challenging, and the result is not pretty.

- The D in the base of the logarithm is of little importance unless clocks are of poor quality. A standard quartz oscillator guarantees that $\vartheta - 1 \leq 10^{-5}$. Even a gigantic diameter of 10^5 would not affect the bound by more than a factor 2 for such clocks!

- The assumption that $\vartheta \leq 2$ in Theorem 3.4 is an artifact of the proof. However, hardware clocks that are this inaccurate hardly deserve the name "clock," so this corner case is not of interest.

- Overall, the GCS algorithm from the previous lecture appears to be optimal or very close to optimal for essentially all choices of parameters.

- Don't fall into the trap of forgetting that relaxing the model enables better solutions! For instance, if it is not important that clocks make progress at all times (or most of the time), constant local skew can be achieved (buzzword: α-synchronizer)!

Bibliographic Notes

There is not much to add to the notes for the previous lecture. The seminal paper by Fan and Lynch [FL06] introducing the problem provided Theorem 3.4. Meier and Thiele show that essentially the same lower bound arises from bounded communication rates, without uncertainty (i.e., $u = 0$) [**?**]. Theorem 3.1 follows [LLW10], which also tightens the lower bound for unbounded clock rates by removing the D from the base of the logarithm. In the dynamic setting, one can show bounds on how quickly an edge can be incorporated into the subgraph of edges that satisfy the skew bounds, and asymptotic optimality can be achieved simultaneously with other guarantees [KLO11, KLLO10].

Bibliography

[FL06] Rui Fan and Nancy Lynch. Gradient Clock Synchronization. *Distributed Computing*, 18(4):255–266, 2006.

[KLLO10] Fabian Kuhn, Christoph Lenzen, Thomas Locher, and Rotem Oshman. Optimal Gradient Clock Synchronization in Dynamic Networks. *CoRR*, abs/1005.2894, 2010.

[KLO11] Fabian Kuhn, Thomas Locher, and Rotem Oshman. Gradient Clock Synchronization in Dynamic Networks. *Theory Comput. Syst.*, 49(4):781–816, 2011.

[LLW10] Christoph Lenzen, Thomas Locher, and Roger Wattenhofer. Tight Bounds for Clock Synchronization. *J. ACM*, 57(2):8:1–8:42, 2010.

Lecture 4

Fault-Tolerant Clock Synchronization

In the previous lectures, we assumed that the world is a happy place without any kind of faults. This is not a realistic assumption in large-scale systems, and it is an issue in high reliability systems as well. After all, if the system clock fails, there may be no further computations at all!

As, in general, it is difficult to predict what kind of faults may happen, again we assume a worst-case model: Failing nodes may behave in *any* conceivable manner, including collusion, predicting the future, sending conflicting information to different nodes, or even pretending to be correct nodes (for a while). In other words, the system should still function no matter what kind of faults may occur. This may be overly pessimistic in the sense that "real" faults might have a very hard time to produce such behavior. However, if we *can* handle all of these possibilities, we're on the safe side in that we do not have to study what kind of faults may actually happen and verify the resulting fault model(s) for each and every system we build.

Definition 4.1 (Byzantine Faults). *A* Byzantine *faulty node may behave arbitrarily, i.e., it does not follow any algorithm described by the system designer. The set of faulty nodes is (initially) unknown to the other nodes. In other words, the algorithm must be designed in such a way that it works correctly regardless of which nodes are faulty. "Working correctly" here means that all requirements and guarantees on clocks, skews, etc. need only be satisfied by the set V_g of nodes that are not faulty.*

Unsurprisingly, such a strong fault model results in limitations on what can be achieved. For instance, if more than half of the nodes in the system are faulty, there is no way to achieve any kind of synchronization. In fact, even if half of the *neighbors* of some node are faulty, this is impossible. The intuition is simple: Split the neighborhood of some node v in two sets A and B and consider two executions, $\mathcal{E}_A$ and $\mathcal{E}_B$, such that A is faulty in $\mathcal{E}_A$ and B is faulty in $\mathcal{E}_B$. Given that A is faulty in $\mathcal{E}_A$, B and v need to stay synchronized in $\mathcal{E}_A$, regardless of what the nodes in A do. However, the same applies to $\mathcal{E}_B$ with the roles of A and B reversed. However, A and B can have different opinions on the time, and v has no way of figuring out which set to trust.

In fact, it turns out that the number f of faulty nodes must satisfy $3f < n$ or no solution is possible (without cryptographic assumptions); we show this later. Motivated by the above considerations, we also confine ourselves to G being a complete graph: each node is connected to each other node, i.e., each pair of nodes can communicate directly.

4.1 The Pulse Synchronization Problem

Let's study a simpler version of the clock synchronization problem, which we call *pulse synchronization.* Instead of outputting a logical clock at all times, nodes merely need to jointly generate roughly synchronized *pulses* whose frequency is bounded from above and below.

Definition 4.2 (Pulse Synchronization)**.** *Each (non-faulty) node is to generate each pulse $i \in \mathbb{N}$ exactly once. Denoting by $p_{v,i}$ the time when node v generates pulse i, we require that there are $\mathcal{S}, P_{\min}, P_{\max} \in \mathbb{R}^+$ so that*

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

Remarks:

- The idea is to interpret the pulses as the "ticks" of a common clock.
- Ideally, $\mathcal{S}$ is as small as possible, while $P_{\min}$ and $P_{\max}$ are as close to each other as possible and can be scaled freely.
- Due to the lower bound from Lecture 1, we have that $\mathcal{S} \geq u/2$.
- Clearly, we cannot expect better than $P_{\max} \geq \vartheta P_{\min}$, i.e., matching the quality of the hardware clocks. Also, $P_{\max} - P_{\min} \geq \mathcal{S}$.
- Because $D = 1$, the problem would be trivial without faults. For instance, the Max Algorithm would achieve skew $u + (\vartheta - 1)(d + T)$, and pulses could be triggered every $\Theta(\mathcal{G})$ local time.
- The difficulty lies in preventing the faulty nodes from dividing the correctly functioning nodes into unsynchronized subsets.

4.2 A Variant of the Srikanth-Toueg Algorithm

One of our design goals here is to keep the algorithm extremely simple. To this end, we decide that

- Nodes will communicate by broadcast (i.e., sending the same information to all other nodes, for simplicity including themselves) only. Note that faulty nodes do not need to stick to this rule!
- Messages are going to be very short. In fact, there is only a single type of message, carrying the information that a node transitioned to state PROPOSE.

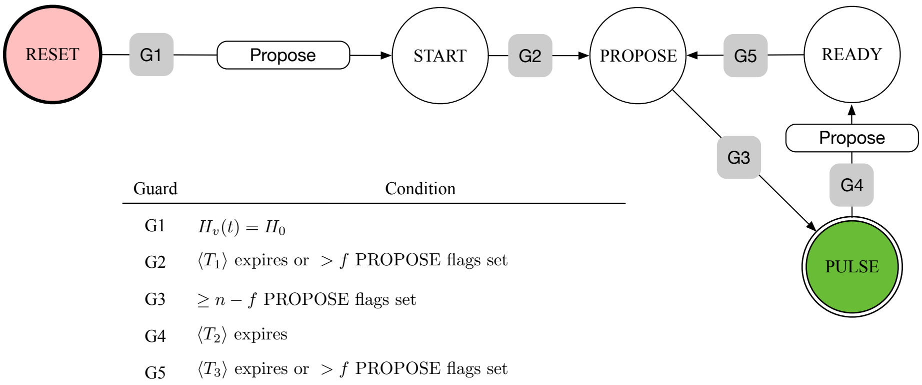
- Nodes will store, for each node, whether they received such a message. On some state transitions, they will reset these memory flags to 0 (i.e., no message received yet).

- Not accounting for the memory flags, each node runs a state machine with a constant number of states.

- Transitions in this state machine are triggered by expressions involving (i) the own state, (ii) thresholds for the number of memory flags that are 1, and (iii) timeouts. A timeout means that a node waits for a certain amount of local time after entering a state before considering a 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 $\max_{v \in V_g}\{H_v(0)\} < H_0$.

The algorithm, from the perspective of a node, is depicted in Figure 4.1. 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.



Guard	Condition
G1	$H_v(t) = H_0$
G2	$\langle T_1 \rangle$ expires or $> f$ PROPOSE flags set
G3	$\geq n - f$ PROPOSE flags set
G4	$\langle T_2 \rangle$ expires
G5	$\langle T_3 \rangle$ expires or $> f$ PROPOSE flags set

Figure 4.1: 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 indicates that a node clears its PROPOSE memory flags when transitioning from RESET to START and 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 (4.1)–(4.4).

- 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 may never send a message, 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:

$$H_0 > \max_{v\in V_g}\{H_v(0)\} \tag{4.1}$$

$$\frac{T_1}{\vartheta} \geq H_0 \tag{4.2}$$

$$\frac{T_2}{\vartheta} \geq 3d \tag{4.3}$$

$$\frac{T_3}{\vartheta} \geq \left(1 - \frac{1}{\vartheta}\right) T_2 + 2d \tag{4.4}$$

Lemma 4.3. *Suppose $3f < n$ 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 $t' \in (t - \Delta + T_1/\vartheta, t + T_1 - d)$ ($t' \in (t - \Delta + T_3/\vartheta, t + T_3 - d)$) 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. Denote by t_p 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.

From this observation we can infer that 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 $[t', t' + 2d)$. □

Theorem 4.4. *Suppose that $3f < n$ and the above constraints are satisfied. Then the algorithm given in Figure 4.1 solves the pulse synchronization problem with $\mathcal{S} = 2d$, $P_{\min} = (T_2 + T_3)/\vartheta - 2d$ 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 4.3. The first time, we use that all nodes start with hardware clock values in the range $[0, H_0)$ by (4.1). As hardware clocks run at least at rate 1, thus all nodes transition to state START by time H_0. By (4.2), 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 $i + 1$, (4.3) 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 (4.4) we can apply Lemma 4.3 with $t = \max_{w \in V_g}\{p_{w,i}\} + T_2$ and $\Delta = (1 - 1/\vartheta)T_2 + 2d$, yielding pulse times $p_{v,i+1}$, $v \in V_g$, satisfying the stated skew bound.

It remains to show that $\min_{v \in V_g}\{p_{v,i+1}\} - \max_{v \in V_g}\{p_{v,i}\} \geq (T_2 + T_3)/\vartheta - 2d$ 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 4.3,

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

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

Remarks:

- The skew bound of $2d$ can be improved to $d+u$ by a more careful analysis; you'll show this as an exercise.
- 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 $P_{\max} = (3\vartheta^2 + 2\vartheta + 2)d$.

4.3 Impossibility of Synchronization for $3f \geq n$

If $3f \geq n$, the faulty nodes can force correct nodes to lose synchronization in some executions. We will use indistinguishability again, but this time there will always be some correct nodes who can see a difference. The issue is that they cannot *prove* to the other correct nodes that it's not them who are faulty.

We partition the node set into three sets $A, B, C \subset V$ so that $|A|, |B|, |C| \leq f$. We will 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. In each execution, one of the sets consists entirely of faulty nodes. In each of the other sets, the hardware clocks of all nodes will be identical. The same holds for the faulty set, but the

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

Table 4.1: Hardware clock speeds in the different executions for the different sets. The red entries indicate faulty sets, simulating a clock speed of $\rho^3 t$ 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.

nodes there play both sides differently: to one set, they make their clocks appear to be very slow, to the other they make them appear fast. All clock rates (actual or simulated) will lie between 1 and ρ^3, where $\rho > 1$ is small enough so that $\rho^3 \leq \vartheta$ and $d \leq \rho^3(d-u)$; this way, message delays can be chosen such that messages arrive at the same local times without violating message delay bounds.

Note that for each pair of consecutive executions, the executions are indistinguishable to the set that is correct in both of them *and* a factor of $\rho > 1$ lies between the speeds of hardware clocks. This means that the pulses are generated at a by factor ρ higher speed. However, as the skew bounds are to be satisfied, this means that also the set of correct nodes that *knows* that something is different will have to generate pulses faster. This means that in execution $\mathcal{E}_i$, pulses are generated at an amortized rate of (at least) $\rho^i P_{\min}$. For $i > \log_\rho P_{\max}/P_{\min}$, this is a contradiction.

Lemma 4.5. *Suppose $3f \geq n$. Then, for any algorithm $\mathcal{A}$, there exists $\rho > 1$ and a sequence of executions $\mathcal{E}_i$, $i \in \mathbb{N}_0$, with the properties stated in Table 4.1.*

Proof. Choose $\rho := \min\left\{\vartheta, \frac{d}{d-u}\right\}^{1/3}$. We construct the entire sequence concurrently, where we advance real time in execution $\mathcal{E}_i$ at speed ρ^{-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 ρ, 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 $t = H_v^{(\mathcal{E}_i)}(t)/\rho$, it sends the same message in $\mathcal{E}_i$ at time t. Thus, it is received at local time

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

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 $t = H_v^{(\mathcal{E}_{i-1})}(t)/\rho^2$, it sends the same message in $\mathcal{E}_i$ at time t/ρ^3. Thus, it is received at local time

$$H_w^{(\mathcal{E}_i)}\left(\frac{t}{\rho^3}\right) + d = \frac{t}{\rho} + d = H_w^{(\mathcal{E}_{i-1})}(t) + d\,.$$

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 ρ^{-i}." Think of each faulty node as simulating two virtual nodes, one for messages sent "to the left," which has local time $\rho^3 t$ 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 ρ: 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 ρt; 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 t/ρ. Thus, when simulating all executions concurrently, where $\mathcal{E}_i$ progresses at rate ρ^{-i}, at all times the behavior of faulty nodes according to the above scheme can be determined. This completes the proof. □

Theorem 4.6. *Pulse synchronization is impossible if $3f \geq n$.*

Proof. Assume for contradiction that there is an algorithm solving pulse synchronization. We apply Lemma 4.5, yielding a sequence of executions $\mathcal{E}_i$ with the properties stated in Table 4.1. We will show that pulses are generated arbitrarily fast, contradicting the minimum period requirement. We show this by induction on i, where the induction hypothesis is that there is some $v \in V_g$ satisfying that

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

for all $j \in \mathbb{N}_0$, where $\rho > 1$ is given by Lemma 4.5. This is trivial for the base case $i = 0$ by the maximum period requirement.

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

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

for all $j \in \mathbb{N}_0$. By Lemma 4.5, w cannot distinguish between $\mathcal{E}_i$ and $\mathcal{E}_{i+1}$. Because $H_w^{(\mathcal{E}_{i+1})}(t/\rho) = \rho t = H_w^{(\mathcal{E}_{i+1})}(t)$, we conclude that $p_{w,j}^{(\mathcal{E}_{i+1})} = \rho^{-1}p_{w,j}^{(\mathcal{E}_i)}$ for all $j \in \mathbb{N}_0$. Hence,

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

for all $j \in \mathbb{N}_0$, completing the induction step.

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

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

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

Bibliographic Notes

The algorithm presented in this lecture is a variant of the Srikanth-Toueg algorithm [ST87]. An actual implementation in hardware [FS12] (of another variant) was performed in the DARTS project. In a form close to the one presented here, it was first given in [DFL$^+$15], a survey on fault-tolerant clocking methods for hardware. In all of these cases, the main difference to the original is getting rid of communicating the "tick" number explicitly. The impossibility of achieving synchronization if $f \geq n/3$ was first shown in [DHS86]. Conceptually, the underlying argument is related to the impossibility of consensus in synchronous systems with $f \geq n/3$ Byzantine faults [PSL80].

Concerning the skew bound, we know that $u/2$ skew cannot be avoided from the first lecture. Moreover, $(1 - 1/\vartheta)d/2$ skew cannot be avoided either, as it takes d time to communicate. Note that the upper bound of $2d$ shown here only holds on the *real* time between corresponding ticks; if we derive continuous logical clocks, we get at least an additional $\Omega((\vartheta - 1)d)$ contribution to the skew from the hardware clock drift in between ticks, so there is no contradiction. We'll push the skew down to a matching $\mathcal{O}(u + (\vartheta - 1)d)$ in the next lecture.

Bibliography

[DFL$^+$15] Danny Dolev, Matthias Függer, Christoph Lenzen, Ulrich Schmid, and Andreas Steininger. Fault-tolerant Distributed Systems in Hardware. *Bulletin of the EATCS*, 116, 2015.

[DHS86] Danny Dolev, Joseph Y. Halpern, and H.Raymond Strong. On the Possibility and Impossibility of Achieving Clock Synchronization. *Journal of Computer and System Sciences*, 32(2):230–250, 1986.

[FS12] Matthias Függer and Ulrich Schmid. Reconciling fault-tolerant distributed computing and systems-on-chip. *Distributed Computing*, 24(6):323–355, 2012.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *J. ACM*, 27(2):228–234, 1980.

[ST87] T. K. Srikanth and Sam Toueg. Optimal Clock Synchronization. *J. ACM*, 34(3):626–645, 1987.

Lecture 5

Synchronizing by Approximate Agreement

In the previous lecture, we've seen how to achieve a skew of $\mathcal{O}(d)$ in a system of n fully connected nodes with $f < n/3$ Byzantine faults. We've also seen that we can't do any better in terms of the number of faults that can be tolerated. So let's ask our usual question: Is this skew bound (asymptotically) optimal or can we do better? Already in a fault-free system, we know that we can't beat $\Omega(u + (\vartheta - 1)d)$. But can this bound be attained in the presence of faults?

5.1 Approximate Agreement

The answer is provided by leveraging techniques for the task of *approximate agreement.* For this problem, we assume the (convenient) abstraction of a synchronously operating system.

Definition 5.1 (Synchronous Execution)**.** *A synchronous execution proceeds in* synchronous rounds. *At the start of the execution, each node receives an input (whose type depends on the task at hand). In each round,*

1. *nodes perform local computations,*
2. *send messages to their neighbors in the network graph,*
3. *receive the messages of their neighbors, and (optionally)*
4. *may compute an output value and terminate (i.e., stop executing the other steps in future rounds).*

Note that a synchronous execution of a deterministic algorithm is fully determined by the input values and the (arbitrary) messages sent by faulty nodes. The key performance measures are the round complexity — the number of rounds until all nodes terminated — and the maximum size of messages (sent by correct nodes).

This model provides a very clean abstraction for describing the tool we would like to use.

Definition 5.2 (Approximate Agreement)**.** *Each node $v \in V$ is given an input value $r_v \in \mathbb{R}$. Given a constant $\varepsilon > 0$, the task is to generate output values $o_v \in \mathbb{R}$ so that*

agreement: $\max_{v,w \in V_g}\{o_v - o_w\} \leq \varepsilon$,

validity: $\forall v \in V_g\colon \min_{w \in V_g}\{r_w\} \leq o_w \leq \max_{w \in V_g}\{r_w\}$, *and*

termination: *each $v \in V_g$ determines is output o_v and terminates within a finite number of rounds.*

Remarks:

- The synchronous model is a highly useful abstraction in distributed computing. With known upper bounds $\mathcal{L}$ on local skew, λ on logical clock rates, and d on message delays, it is straightforward to simulate. Assuming that $\max_{v \in V_g}\{L_v(0)\} = L$, nodes send their messages for round $r \in \mathbb{N}$ at the time t when $L_v(t) = L + (r-1)\lambda(d + \mathcal{L})$. Thus, all messages for round r are received before the ones for round $r+1$ need to be sent.

- If the round number is not to be sent along with the message or for some other reason it's important that messages for round $r+1$ must not arrive anywhere before round r is complete at all nodes, one may add an additional $\lambda\mathcal{S}$ at the beginning of the round before messages are sent. We will use this in our algorithm!

- Recall that d accounts for local computations, not only the time messages are in transit. Thus, involved calculations affect the time the simulation takes via d!

- Lower bounds on the progress of logical clocks are needed for guaranteeing progress. The better the lower bound, the earlier the simulation completes (i.e., all nodes terminate).

- Without faults, synchronizers provide elegant solutions that work even if d is unknown. However, synchronizers wait for proof that all other nodes finished their current round before proceeding. Even a single crash fault (a node not sending any messages any more) would halt the entire system!

- Once we solved approximate agreement in this abstract model, we will employ it to agree on when the nodes should generate clock pulses, i.e., solve the pulse synchronization problem with it.

- The simulation of the synchronous algorithm and maintaining a small skew will go hand in hand!

Solving Approximate Agreement

Definition 5.3 (Diameters of Vectors)**.** *Denote by $\vec{r}$ the $|V_g|$-dimensional vector of correct nodes' inputs, i.e., $(\vec{r})_v = r_v$ for $v \in V_g$. Denote by $r^{(k)}$, $k \in \{1, \dots, |V_g|\}$, the k^{th} entry when ordering the entries of $\vec{r}$ ascendingly.*

Algorithm 5.1: Approximate agreement step at node $v \in V_g$ (with synchronous message exchange).

```
// node $v$ is given input value $r_v$;
broadcast $r_v$ to all nodes (including self);
receive $\hat{r}_{wv}$ from each node $w$ ($\hat{r}_{wv} := r_v$ if no message with correct type
  of content from $w$ received);
$S_v \leftarrow \{\hat{r}_{wv} \,|\, w \in V\}$;
$o_v \leftarrow \frac{S_v^{(f+1)} + S_v^{(n-f)}}{2}$;
return $o_v$;
```

The diameter $\|\vec{r}\|$ *of $\vec{r}$ is the difference between the maximum and minimum components of $\vec{r}$. Formally,*

$$\|\vec{r}\| := r^{(|V_g|)} - r^{(1)} = \max_{r \in V_g}\{r_v\} - \min_{v \in V_g}\{r_v\}.$$

We will use the same notation for other values, e.g. $\vec{o}$, $o^{(k)}$, $\|\vec{o}\|$, etc.

For simplicity, we assume that $|V_g| = n - f$ in the following; all statements can be adapted by replacing $n - f$ with $|V_g|$ where appropriate. As usual, we require that $3f < n$.

Intuitively, Algorithm 5.1 discards the smallest and largest f values each to ensure that values from faulty nodes cannot cause outputs to lie outside the range spanned by the correct nodes' values. Afterwards, o_v is determined as the midpoint of the interval spanned by the remaining values. Since $f < n/3$, i.e., $n - f \geq 2f + 1$, the median of correct nodes' values is part of all intervals computed by correct nodes. From this, it is easy to see that $\|\vec{o}\| \leq \|\vec{r}\|/2$. We now prove these properties.

Lemma 5.4.

$$\forall v \in V_g \colon r^{(1)} \leq o_v \leq r^{(n-f)}\,.$$

Proof. As there are at most f faulty nodes, for $v \in V_g$ we have that

$$S_v^{(f+1)} \geq \min_{w \in V_g}\{\hat{r}_{wv}\} = r^{(1)}\,.$$

Analogously, $S_v^{(n-f)} \leq r^{(n-f)}$. We conclude that

$$r^{(1)} \leq S_v^{(f+1)} \leq \frac{S_v^{(f+1)} + S_v^{(n-f)}}{2} = o_v \leq S_v^{(n-f)} \leq r^{(n-f)}\,. \qquad \square$$

Lemma 5.5. $\|\vec{o}\| \leq \|\vec{r}\|/2$.

Proof. Since $f < n/3$, we have that $n - f \geq 2f + 1$. Hence, for all $v \in V_g$,

$$r^{(1)} \leq S_v^{(f+1)} \leq r^{(f+1)} \leq S_v^{(2f+1)} \leq S_v^{(n-f)} \leq r^{(n-f)}\,.$$

For any $v, w \in V_g$, it follows that

$$\begin{aligned} o_v - o_w &= \frac{S_v^{(f+1)} - S_w^{(f+1)} + S_v^{(n-f)} - S_w^{(n-f)}}{2} \\ &\leq \frac{r^{(f+1)} - r^{(1)} + r^{(n-f)} - r^{(f+1)}}{2} = \frac{r^{(n-f)} - r^{(1)}}{2} \\ &= \frac{\|\vec{r}\|}{2} \,. \end{aligned}$$

As $v, w \in V_g$ were arbitrary, this yields $\|\vec{o}\| \leq \|\vec{r}\|/2$. □

Applying this approach inductively yields a straightforward algorithm provided an upper bound $R \geq r^{(|V_g|)} - r^{(1)}$ is known.

Theorem 5.6 (Approximate Agreement)**.** *Applying Algorithm 5.1 iteratively (using the output of one step as input to the next) for $\lceil \log(R/\varepsilon) \rceil$ steps solves approximate agreement.*

Proof. Applying Lemma 5.5 inductively shows agreement. Applying Lemma 5.4 inductively shows validity. By construction, all nodes terminate after $\lceil \log(R/\varepsilon) \rceil$ synchronous rounds. □

Modifications for the Pulse Synchronization Problem

In our setting, we will not be able to guarantee exact communication of clock values. Accordingly, we slightly modify the communication model. More specifically, at certain times, nodes will need estimates of each other's logical clock values. Node v will use its estimate of w's clock value as approximation of the "input" r_w of $w \in V$. Thus, instead of receiving $\hat{r}_{wv} = r_w$ from $w \in V$, v will receive

$$r_w - \delta < \hat{r}_{wv} \leq r_w \,.$$

As shifting the values $\hat{r}_{wv}$ in Algorithm 5.1 by less than δ will affect the outputs by less than δ, we obtain the following corollary to Lemmas 5.4 and 5.5. See Figure 5.1 for a visualization.

Corollary 5.7. *With the above modification to the communication model, Algorithm 5.1 guarantees*

(i) $\forall v \in V_g \colon r^{(1)} - \delta < o_v \leq r^{(n-f)}$ *and*

(ii) $\|\vec{o}\| \leq \|\vec{r}\|/2 + \delta$.

Remarks:

- Now all we need to do is to gather estimates, use Algorithm 5.1 to determine adjustments to the logical clocks, and iterate.
- Trivia: When I suggested to Danny Dolev that one could make use of approximate agreement as the basis for a clock synchronization algorithm, he told me that this was precisely the motivation for introducing the problem and pointed me towards the paper implementing this approach. He and his co-authors were merely about three decades and a brilliant abstraction ahead of me!

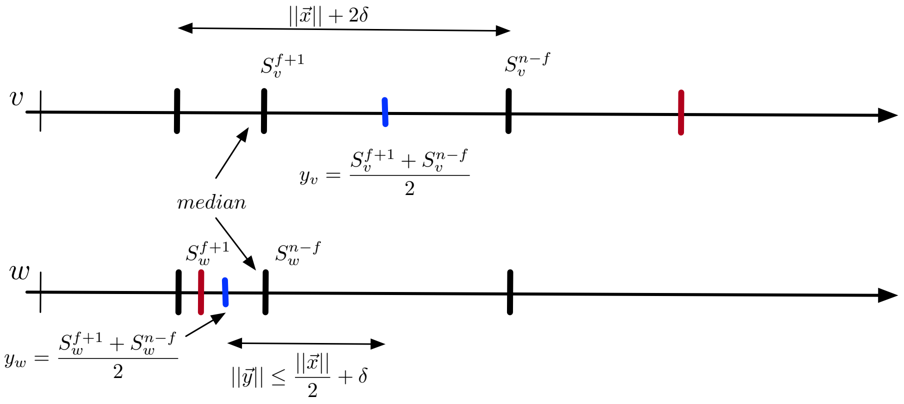

$||\vec{x}|| + 2\delta$
S_v^{f+1}
S_v^{n-f}
v
$y_v = \frac{S_v^{f+1} + S_v^{n-f}}{2}$
median
S_w^{f+1}
S_w^{n-f}
w
$y_w = \frac{S_w^{f+1} + S_w^{n-f}}{2}$
$||\vec{y}|| \leq \frac{||\vec{x}||}{2} + \delta$

Figure 5.1: An execution of Algorithm 5.1 at nodes v and w of a system consisting of $n = 4$ nodes. There is a single faulty node and its values are indicated in red. Note that the ranges spanned by the values received from non-faulty nodes are *almost* identical; the difference originates in the perturbations of up to δ.

5.2 A Variant of the Lynch-Welch Algorithm

The algorithm is now constructed as follows. Assuming some bound $H \geq \max_{v \in V_g}\{H_v(0)\}$ on the skew at initialization, nodes generate their first pulse at local time H. This marks the (local) start of the first round. Then they wait until they can be sure that all nodes have generated their pulse. At the respective hardware time, they transmit an empty message — no content is needed, as the local time when the message is sent is hardwired into the algorithm. Then

Algorithm 5.2: Lynch-Welch pulse synchronization algorithm, code for node $v \in V_g$. $\mathcal{S}$ denotes a (to-be-determined) upper bound on $\|\vec{p}_r\|$ for each $r \in \mathbb{N}$ and T is the nominal round duration.

```
// $H_w(0) \in [0, \mathcal{S})$ for all $w \in V$
set $L_v(0) := H_v(0)$
increase $L_v$ at rate $h_v$ at all times
generate pulse 1 at the time $p_{v,1}$ with $L_v(p_{v,1}) = \mathcal{S}$;
**foreach** *round* $r \in \mathbb{N}$ **do**
    wait until local time $(r-1)T + (\vartheta + 1)\mathcal{S}$;// all nodes are in round $r$
    broadcast empty message to all nodes (including self);
    wait until time $\tau_{v,r}$ when $L_v(\tau_{v,r}) = (r-1)T + (\vartheta^2 + \vartheta + 1)\mathcal{S} + \vartheta d$;
      // correct nodes' messages arrived
    **for** *each node* $w \in V$ **do**
        // abbreviate $p_r := \max_{w \in V_g}\{p_{w,r}\}$ (unknown to the node!)
        compute $\Delta^v_w \in (L_v(p_r) - L_w(p_r) - \delta, L_v(p_r) - L_w(p_r)]$
    $S_v \leftarrow \{\Delta^v_w \,|\, w \in V\}$ (as multiset, i.e., values may repeat)
    $L_v(\tau_{v,r}) \leftarrow L_v(\tau_{v,r}) + \left(S_v^{(f+1)} + S_v^{(n-f)}\right)/2$
    generate pulse $r+1$ at the time $p_{v,r+1}$ with $L_v(p_{v,r+1}) = \mathcal{S} + rT$
```

nodes wait until the local time when all such messages from correct nodes are certainly received and compute their estimates of the relative clock differences to other nodes. Finally, they apply Algorithm 5.1 to compute an adjustment to the (local) starting time of the next round. This ensures bounded skew for the next pulse and thus also the starting times of the next round. From there, the process is iterated.

Algorithm 5.1 is phrased in a parametrized fashion suitable for the analysis. This means that we assume a skew bound of $\mathcal{S}$ to hold on initialization, an error bound δ on the logical clock estimates nodes compute of each other, and a nominal round duration of T. We then determine valid choices for these parameters from the analysis, where we need to determine δ depending on how the estimates are computed.

"Rounds" of the algorithm simulate the synchronous operation assumed in the approximate agreement problem, where each iteration of the loop simulates one synchronous round. For this to work as intended, two requirements need to be met in each round:

(i) Messages sent by correct nodes are received at all correct nodes after starting the round and before they compute their clock adjustment, i.e., during $[p_{v,r}, \tau_{v,r}]$.

(ii) T is large enough to ensure that the clock adjustment makes no logical clock "jump" past $L_v(p_{v,r+1}) = \mathcal{S} + rT$, skipping a pulse.

If these properties are satisfied in round r, we will say that *round r is executed correctly.* We will show that this holds for all $r \in \mathbb{N}$ inductively, where the induction hypothesis is that $\|\vec{p}_r\| \leq \mathcal{S}$; this simulatenously shows that the algorithm has a small skew! For $r = 1$, this is immediate from our assumption on the initial hardware clock values.

Lemma 5.8. *Suppose that $T/\vartheta \geq (\vartheta^2+\vartheta+1)\mathcal{S}+\vartheta d$ and $\mathcal{S} \geq 2(\delta+(1-1/\vartheta)T)$. Moreover, assume that for $r \in \mathbb{N}$ it holds that all prior rounds have been executed correctly, and that $\|\vec{p}_r\| \leq \mathcal{S}$. Then*

(i) round r is executed correctly,

(ii) $\|\vec{p}_{r+1}\| \leq \mathcal{S}$, and

(iii) $T/\vartheta - \mathcal{S} \leq p_{r+1} - p_r \leq T + \delta$.

Proof. By assumption, no messages sent by correct nodes in rounds $r' < r$ are received in round r. Consider the message $v \in V$ sends after entering round r. It is sent no earlier than time $p_{v,r} + \mathcal{S} \geq \max_{w \in V_g}\{p_{w,r}\}$, as $\|\vec{p}_r\| \leq \mathcal{S}$ by assumption. It is received before time

$$p_{v,r} + \vartheta\mathcal{S} + d \leq \min_{w \in V_g}\{p_{w,r}\} + (\vartheta+1)\mathcal{S} + d\,.$$

As $\tau_{w,r} \geq p_{w,r} + (\vartheta+1)\mathcal{S} + d$ for all $w \in V_g$, this shows part (i) of correct execution of round r.

Concerning part (ii), we apply statement (i) of Corollary 5.7, showing that the logical clock of $v \in V_g$ cannot be set to a larger value than

$$\begin{aligned} L_v(\tau_{v,r}) &\leq L_v(p_r) + \int_{p_r}^{\tau_{v,r}} h_v(t)\,dt - \min_{w \in V_g}\{\Delta_w^v\} \\ &\leq L_v(p_r) + \int_{p_r}^{\tau_{v,r}} h_v(t)\,dt + \max_{w \in V_g}\{L_w(p_r)\} - L_v(p_r) \\ &\leq \max_{w \in V_g}\{L_w(p_r)\} + \vartheta(\tau_{v,r} - p_r) \\ &\leq \max_{w \in V_g}\{L_w(p_{w,r}) + \vartheta(\tau_{v,r} - p_{w,r})\} \\ &= (r-1)T + \mathcal{S} + \vartheta\left(\tau_{v,r} - \min_{w \in V_g}\{p_{w,r}\}\right). \end{aligned}$$

It follows that no node can reach logical clock value $rT + \mathcal{S}$ earlier than time $\min_{w \in V_g}\{p_{w,r}\} + T/\vartheta$. In particular, this is bounded from below by $p_r + T/\vartheta - \mathcal{S}$, showing the lower bound of the third claim of the lemma.

On the other hand, for all $v \in V_g$, we have that

$$\tau_{v,r} \leq p_{v,r} + (\vartheta^2 + \vartheta)\mathcal{S} + \vartheta d \leq \min_{w \in V_g}\{p_{w,r}) + (\vartheta^2 + \vartheta + 1)\mathcal{S} + \vartheta d\,,$$

where the second step uses that $\|p_{v,r}\| \leq \mathcal{S}$. As $T/\vartheta \geq (\vartheta^2 + \vartheta + 1)\mathcal{S} + \vartheta d$, this shows that round r is executed correctly. In particular, the times $p_{v,r+1}$, $v \in V_g$, are well-defined.

By statement (i) of Corollary 5.7, we have that, at time $\tau_{v,r}$, $v \in V_g$ cannot set its logical clock to a smaller value than

$$\begin{aligned} L_v(\tau_{v,r}) &\geq L_v(p_r) + \int_{p_r}^{\tau_{v,r}} h_v(t)\,dt + \min_{w \in V_g}\{L_w(p_r)\} - L_v(p_r) - \delta \\ &\geq \int_{p_r}^{\tau_{v,r}} h_v(t)\,dt + \min_{w \in V_g}\{L_w(p_r)\} - \delta \\ &= \int_{p_r}^{\tau_{v,r}} h_v(t)\,dt + (r-1)T + \mathcal{S} - \delta\,. \end{aligned}$$

As hardware clock rates are at least 1, this shows that $p_{r+1} \leq p_r + T + \delta$, i.e., the upper bound of the third claim of the lemma holds.

It remains to show the second claim, i.e., the bound on the skew. To simplify our reasoning, pretend that the clock adjustments from round r would take place at time p_r. Denote by $L'_v(p_r)$, $v \in V_g$, the respective modified logical clocks, which increase at the rate of the hardware clocks during round r and satisfy $L'_v(t) = L_v(t)$ at times $t \geq \tau_{v,r}$. By the above bound, we thus have that

$$L'_v(p_r) = L_v(\tau_{v,r}) - \int_{p_r}^{\tau_{v,r}} h_v(t)\,dt \geq (r-1)T + \mathcal{S} - \delta\,.$$

Next, note that $\|\vec{L}(p_r)\| \leq \vartheta\|\vec{p_r}\| \leq \vartheta\mathcal{S}$ by assumption. By statement (ii) of Corollary 5.7, this implies that $\|\vec{L}'(p_r)\| \leq \vartheta\mathcal{S}/2 + \delta$. Now let $v, w \in V_g$ maximize $p_{r+1,v} - p_{r+1,w}$. We have that $p_{r+1,v} - p_r \leq rT + \mathcal{S} - L'_v(p_r)$ and $p_{r+1,w} - p_r \geq (rT + \mathcal{S} - L'_w(p_r))/\vartheta$ due to the bounds on the hardware clock rates. Hence,

$$\begin{aligned}
p_{r+1,v} - p_{r+1,w} &\leq L'_w(p_r) - L'_v(p_r) + \left(1 - \frac{1}{\vartheta}\right)(rT + \mathcal{S} - L'_w(p_r))\\
&\leq \|\vec{L}'(p_r)\| + \left(1 - \frac{1}{\vartheta}\right)(rT + \mathcal{S} - (L'_v(p_r) + \|\vec{L}'(p_r)\|))\\
&\leq \|\vec{L}'(p_r)\| + \left(1 - \frac{1}{\vartheta}\right)(T + \delta - \|\vec{L}'(p_r)\|)\\
&\leq \frac{\vartheta\mathcal{S}}{2} + \delta + \left(1 - \frac{1}{\vartheta}\right)\left(T - \frac{\vartheta\mathcal{S}}{2}\right)\\
&= \frac{\mathcal{S}}{2} + \delta + \left(1 - \frac{1}{\vartheta}\right)T\,.
\end{aligned}$$

This being bounded by $\mathcal{S}$ is equivalent to $\mathcal{S} \geq 2(\delta + (1 - 1/\vartheta)T)$. □

Before we can prove our main theorem, we need to get a hold on δ. This is a straightforward calculation.

Lemma 5.9. *Suppose round r is executed correctly and $v \in V_g$ receives the message from $w \in V_g$ for this round at time t. Then setting*

$$\Delta_w^v \coloneqq L_v(t) - (r-1)T - (\vartheta^2 + 1)\mathcal{S} - \vartheta d$$

yields an estimate satisfying $\delta \leq u + (\vartheta - 1)d + 2(\vartheta^2 - \vartheta)\mathcal{S}$.

Proof. Denote by t the time when v receives the message from w and by t_s the time when it was sent. We have that

$$L_v(t) - L_w(t_s) \in (L_v(t_s) - L_w(t_s) + d - u, L_v(t_s) - L_w(t_s) + \vartheta d)\,.$$

Moreover,

$$|L_v(t_s) - L_v(p_r) - (L_w(t_s) - L_w(p_r))| \leq (\vartheta - 1)(t_s - p_r) \leq \left(\vartheta^2 - \vartheta\right)\mathcal{S}\,.$$

We conclude that

$$\begin{gathered}
L_v(t) - L_w(t_s) \in\\
(L_v(p_r) - L_w(p_r) + d - u - (\vartheta^2 - \vartheta)\mathcal{S}, L_v(p_r) - L_w(p_r) + \vartheta d + (\vartheta^2 - \vartheta)\mathcal{S})\,.
\end{gathered}$$

As $L_w(t_s) = (r-1)T + (\vartheta + 1)\mathcal{S}$ by the design of the algorithm, the claim of the lemma follows. □

Theorem 5.10. *Assume that $3 + 4\vartheta - 4\vartheta^2 - 2\vartheta^3 > 0$ and that estimates are computed according to Lemma 5.9. For any choice of*

$$T \geq \frac{6\vartheta^4(u+d)}{3 + 4\vartheta - 4\vartheta^2 - 2\vartheta^3} \in \mathcal{O}(d)\,,$$

set

$$\mathcal{S} \coloneqq \frac{2(u + (\vartheta - 1)d + (1 - 1/\vartheta)T)}{1 + 4\vartheta - 4\vartheta^2} \in \mathcal{O}\left(u + \left(1 - \frac{1}{\vartheta}\right)T\right).$$

If $\max_{v \in V}\{H_v(0)\} \leq \mathcal{S}$, then Algorithm 5 solves pulse synchronization with skew $\mathcal{S}$, $P_{\min} \geq T/\vartheta - \mathcal{S}$, and $P_{\max} \leq T + 2\mathcal{S}$.

Proof. Set $\delta := u + (\vartheta - 1)d + 2(\vartheta^2 - \vartheta)\mathcal{S}$ in accordance with Lemma 5.9. Thus,

$$\mathcal{S} = 2\left(u + (\vartheta - 1)d + 2(\vartheta^2 - \vartheta)\mathcal{S} + \left(1 - \frac{1}{\vartheta}\right)T\right) = 2\left(\delta + \left(1 - \frac{1}{\vartheta}\right)T\right).$$

Moreover,

$$T \geq \frac{6\vartheta^4(u+d) + 2(\vartheta^3 - 1)T}{3 + 4\vartheta - 4\vartheta^2 - 2\vartheta^3 + 2(\vartheta^3 - 1)} > \frac{6\vartheta^3(u + (\vartheta - 1)d) + 2(\vartheta^3 - 1)T}{1 + 4\vartheta - 4\vartheta^2} + \vartheta^2 d,$$

i.e.,

$$\frac{T}{\vartheta} > (\vartheta^2 + \vartheta + 1) \cdot \frac{2(u + (\vartheta - 1)d) + 2(1 - 1/\vartheta)T}{1 + 4\vartheta - 4\vartheta^2} + \vartheta d = (\vartheta^2 + \vartheta + 1)\mathcal{S} + \vartheta d.$$

The claim is now shown by a straightforward induction on the pulse number, where the hypothesis includes that all previous rounds have been executed correctly. The induction is anchored at the first pulse, which satisfies the skew bounds due to the assumed bound on the hardware clock values at time 0. The induction step is performed by invoking Lemma 5.8, where Lemma 5.9 shows that δ is indeed a bound on the quality of estimates. We obtain that $\mathcal{S}$ is a bound on the skew for all pulses and that $T/\vartheta - \mathcal{S} \leq p_{r+1} - p_r \leq T + \delta$ for each $r \in \mathbb{N}$. This implies that $P_{\min} \geq (T - \mathcal{S})/\vartheta$ and, using that $\max_{v \in V_g}\{p_{v,r}\} - \min_{v \in V_g}\{p_{v,r}\} \leq \mathcal{S}$ and $\delta < \mathcal{S}$, that $P_{\max} \leq T + 2\mathcal{S}$. □

Remarks:

- The theorem requires that $3 + 4\vartheta - 4\vartheta^2 - 2\vartheta^3 > 0$, which is the case for $\vartheta \leq 1.09$. As ϑ approaches this threshold, the skew goes to ∞.

- Sending $(T, \vartheta) \to (\infty, 1)$, the ratio $P_{\max}/P_{\min} \in (1 + o(1))\vartheta$. However, when sending $T \to \infty$ while keeping ϑ fixed, the ratio converges to a constant $c \in 1 + \mathcal{O}(\vartheta - 1)$.

- If on initialization such a tight skew bound cannot be guaranteed, one can choose T accordingly larger.

- Alternatively, one can only initially use the larger T and keep reducing T alongside the decrease in (the worst-case bound on) the skew. You'll analyze this in the exercises.

- A known bound on the initial skew is necessary for executing the algorithm. You'll show this in the exercises as well.

- We haven't clarified how nodes compute their estimates of faulty nodes' clocks. What if these nodes send no or many messages during a round? The answer is simple: It doesn't matter. As the approximate agreement algorithm works regardless of what values faulty nodes provide, choosing *any* default value for nodes clearly not obeying the protocol will do.

Bibliographic Notes

Approximate agreement was introduced by Dolev et al. [DLP$^+$86], actually having the goal in mind to use it for clock synchronization. As shown by Fekete [Fek86], the rate of convergence provided by their algorithm is close to being asymptotically optimal, and it is asymptotically optimal if only one round of communication per iteration is performed. He also shows that faster convergence is possible if the (maximum) number of possible faults is smaller.

The clock synchronization protocol by Lynch and Welch [LL84] is able to exploit this, too, to achieve faster convergence and thus slightly smaller skews. The respective modification is straightforward and can also be applied to the variant presented in this lecture, which follows [**?**]. The main difference to [LL84] is, just as for the Srikanth-Toueg algorithm from the previous lecture, that no tick numbers are communicated by the algorithm. One can also adjust clock rates (as opposed to just correcting clock offsets), but this requires the additional assumption that hardware clock rates change slowly [KL16]. The Lynch-Welch algorithm has already found its way into practice: it's the synchronization mechanism underlying industrial systems used, e.g., in cars and planes [KB03, FAS09].

Bibliography

[DLP$^+$86] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching Approximate Agreement in the Presence of Faults. *J. ACM*, 33(3):499–516, 1986.

[FAS09] Matthias Függer, Eric Armengaud, and Andreas Steininger. Safely Stimulating the Clock Synchronization Algorithm in Time-Triggered Systems - a Combined Formal & Experimental Approach. *IEEE Trans. Industrial Informatics*, 5(2):132–146, 2009.

[Fek86] A. D. Fekete. Asymptotically Optimal Algorithms for Approximate Agreement. In *Proc. 5th Symposium on Principles of Distributed Computing (PODC)*, pages 73–87, 1986.

[KB03] Hermann Kopetz and G. Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[KL16] Pankaj Khanchandani and Christoph Lenzen. Self-stabilizing Byzantine Clock Synchronization with Optimal Precision. In *Proc. 18th Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 213–230, 2016.

[LL84] Jennifer Lundelius and Nancy Lynch. An Upper and Lower Bound for Clock Synchronization. *Information and Control*, 1984.

Lecture 6

Metastability

In the previous lecture, we've seen how to handle the maximum possible number of worst-case faults (with asymptotically optimal skew bounds!). Or have we? There are fault models that are worse than "just" Byzantine faults. One of the issues that may arise when dealing with low-level hardware implementations of synchronization algorithms is *metastability.* Metastability occurs when a storage element — e.g. a flip-flop — is brought into an unstable equilibrium state. Not binary 0 or binary 1 (low or high output voltage, respectively), but somewhere in between! With the "right" bad input, this is always possible. Figure 6.1 shows how a flip-flop's output responds to critical input signals.

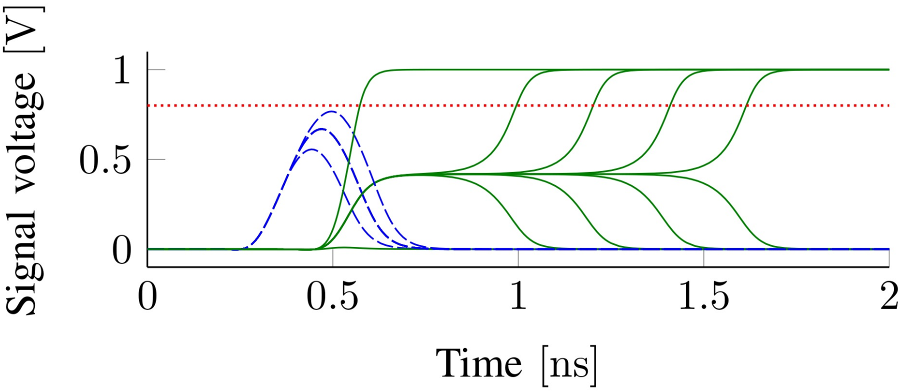


Figure 6.1: Several input (blue) and corresponding output (green) signal traces of a flip-flop. The dotted red line marks the threshold above which the output signal is reliably interpreted as a logical 1. The center blue line is actually not a single one, but many only slightly differing traces, which result in the various green outputs that remain metastable for some time.

Metastability breaks our standard Boolean abstraction, resulting in "faulty" behavior — or rather behavior that is unexpected if we neglect to account for the potential for metastability in our model. Worse, when a metastable flip-flop's output is used in computations and the result is stored in another flip-flop, the latter flip-flop may become metastable as well. So metastability can spread to other parts of the computational logic and “infect” other storage elements.

The reason why we usually neglect metastability is because it’s dealt with by the electrical engineers. As unstable equilibrium state, the probability for sustained metastability decreases exponentially with time. Even the tiniest deviation from the “perfect balance” (e.g. due to thermal noise) gets amplified exponentially, resulting in quick stabilization of the storage element to one of its stable states. Whereever the danger of metastability exists, *synchronizers* are employed, i.e., storage elements specifically designed to resolve metastability as fast as possible, to reduce the probability of metastability sufficiently far before using the registers’ content in computations.

Unfortunately, we want to synchronize clocks as accurately as possible, meaning that we cannot always afford to wait. Making things worse, our “worst-case” fault model of Byzantine nodes does not take into account that Byzantine nodes could try to “infect” correct nodes with metastability. Do we always need to wait for synchronizers to do their job? Can we only guarantee correct operation probabilistically?

Remarks:

- Don’t mistake the synchronizers here with Awerbuch-Sipster network synchronizers. The former deal with metastability, the latter simulate a synchronous network on top of an asynchronous (fault-free) one.

- Using synchronizers is perfectly fine in most applications. Only if we have to respond very quickly (a few nanoseconds or less) to events, we need to look for alternatives.

6.1 Kleene Logic and Circuits

In order to capture how circuits behave in face of metastability, we need to understand how it propagates through logic gates. A very natural way of expressing worst-case behavior of standard logic is Kleene logic. We extend the truth tables for Boolean logic by adding a third logic value M representing metastability and, in fact, *any* signal behavior that is not conform with what we consider a stable 0 or stable 1.

And	0	1
0	0	0
1	0	1

And$_{\mathrm{M}}$	0	1	M
0	0	0	0
1	0	1	M
M	0	M	M

Or	0	1
0	0	1
1	1	1

Or$_{\mathrm{M}}$	0	1	M
0	0	1	M
1	1	1	1
M	M	1	M

Table 6.1: Gate behavior under metastability corresponds to Kleene’s 3-valued logic. A Not$_{\mathrm{M}}$ gate simply maps M to M.

Note carefully that if one stable input already determines the output of a gate, then the other input being M does not matter. This is called *logical masking,* and it is the best guarantee one can hope for: when changing inputs affect the output, one can always “adjust” the input precisely to hitting the spot where the output is neither a logical 0 nor a logical 1.

A definition that extends this desirable behavior to arbitrary Boolean functions is the following.

Definition 6.1 (Metastable Closure). *For $x, y \in \{0,1,\mathrm{M}\}^n$, we say that $x \preceq y$ if and only if $x_i \neq \mathrm{M} \Rightarrow y_i = x_i$ for all $i \in \{1,\dots,n\}$, i.e.,* y is a stabilization *of x. For any Boolean $f\colon \{0,1\}^n \to \{0,1\}^m$, define the* metastable closure f_{M} *of f by*

$$(f_{\mathrm{M}})_i(x) := \begin{cases} 0 & \textit{if } f(y) = 0 \textit{ for all } x \preceq y \in \{0,1\}^n \\ 1 & \textit{if } f(y) = 1 \textit{ for all } x \preceq y \in \{0,1\}^n \\ \mathrm{M} & \textit{else.} \end{cases}$$

Example 6.2 ($\mathrm{MUX}_{\mathrm{M}}$). *A* multiplexer *(or short* MUX*) selects between two input bits based on a select bit (it's third input). Formally,*

$$\mathrm{MUX}\colon \{0,1\}^3 \to \{0,1\}$$
$$\mathrm{MUX}(a,b,s) := \begin{cases} a & \textit{if } s = 0 \\ b & \textit{if } s = 1\,. \end{cases}$$

In Figure 6.2, a standard circuit implementation of a MUX *is shown — and why it does not implement* $\mathrm{MUX}_{\mathrm{M}}$.

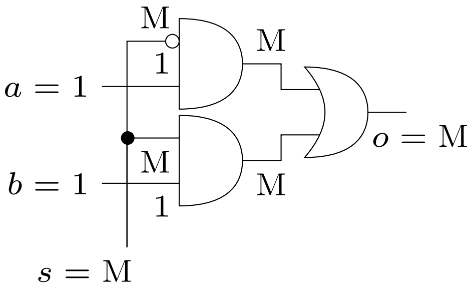


Figure 6.2: Standard MUX implementation. A black dot means that wires are joined (while regular crossings imply no contact). An empty dot is a negation, i.e., a Not gate. And gates are represented by the shapes that are approximately half circles, while the crescent-shaped symbol stands for an Or gate. The figure indicates in which gate inputs and outputs inputs $a = 1$, $b = 1$, and $s = \mathrm{M}$ to the circuit result; note that $\mathrm{MUX}_{\mathrm{M}}(1,1,\mathrm{M}) = 1$.

A metastability-containing multiplexer *(or short* CMUX*) has* $\mathrm{MUX}_{\mathrm{M}}$ *as output function, i.e.,*

$$\mathrm{CMUX}\colon \{0,1,\mathrm{M}\}^3 \to \{0,1,\mathrm{M}\}$$
$$\mathrm{CMUX}(a,b,s) := \mathrm{MUX}_{\mathrm{M}}(a,b,s) = \begin{cases} a & \textit{if } s = 0 \textit{ or } a = b = 0 \\ b & \textit{if } s = 1 \textit{ or } a = b = 1 \\ \mathrm{M} & \textit{else.} \end{cases}$$

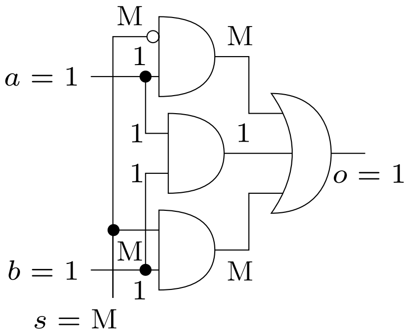


Figure 6.3: CMUX implementation. The figure indicates in which gate inputs and outputs inputs $a = 1$, $b = 1$, and $s = \mathrm{M}$ to the circuit result; the additional AND gate makes sure that the OR gate receives a stable 1 as third input if $a = b = 1$, guaranteeing a stable 1 as output.

As we will see shortly, asking for implementing the metastable closure of a Boolean function is the best we can hope for in a mathematically precise sense. However, we first need to clarify what we mean by "implement." It's exactly what one might expect (see the example above), but a somewhat wordy formalization is necessary to correctly describe the process.

Definition 6.3 (Circuit Behavior). *A combinational circuit C is described as a directed acyclic graph (DAG), where n nodes are marked as inputs and m nodes are marked as outputs. Input nodes have indegree 0, output nodes have indegree 1, and all remaining nodes are gates. A gate implements $f_{\mathrm{M}}\colon \{0,1,\mathrm{M}\}^k \to \{0,1,\mathrm{M}\}$ for a Boolean function $f\colon \{0,1\}^k \to \{0,1\}$. The basic available gates are $\textsc{Or}_{\mathrm{M}}$, $\textsc{And}_{\mathrm{M}}$, $\textsc{Not}_{\mathrm{M}}$, and the constant gates (i.e., no-input gates that provide outputs 0 or 1, respectively). In the DAG, input nodes and gates may have any number of outgoing edges, while output nodes have none. Gates have the number of inputs prescribed by their gate function.*

For a given input $x \in \{0,1,\mathrm{M}\}^n$, the evaluation $C(x)$ of C on x is determined by structural induction as follows. The i^{th} input node evaluates to x_i. As the circuit is described as a DAG, there must be a node for which all incoming edges come from nodes whose evaluation is already determined. If the node is a gate, we apply the gate function to determine its evaluation. If it is an ouput node, it evaluates to the evaluation of the unique node at which its incoming edge originates. This process is iterated until all nodes' evaluation is determined. The output of the circuit is then given by the output nodes' evaluation. We say that C implements *$g : \{0,1,\mathrm{M}\}^n \to \{0,1,\mathrm{M}\}^m$ if $g(x) \preceq C(x)$ for all $x \in \{0,1,\mathrm{M}\}^m$, i.e., $C(x)$ is a stabilization of $g(x)$.*

Remarks:

- The model does not allow for M to stabilize to 0 or 1. This is a worst-case assumption — stable values are never worse than M.

- Accordingly, accepting stabilizations of the desired output from the circuit is fine — the circuit then does better than we ask it to.

6.2 The Limits of Metastability-Containment

Theorem 6.4. *Suppose for a circuit C and a Boolean function f it holds that $C(x) = f(x)$ for all $x \in \{0,1\}^n$. Then $C(x) \preceq f_{\mathrm{M}}(x)$ for all $x \in \{0,1,\mathrm{M}\}^n$.*

Proof. Assume w.l.o.g. that $m = 1$ (otherwise, simply repeat the reasoning for each output bit). We need to show that $C(x) = b \in \{0,1\}$ implies that $f_{\mathrm{M}}(x) = b$. Hence, assume for contradiction that $C(x) = b \in \{0,1\}$, but $f_{\mathrm{M}}(x) \neq b$. Thus, there is $y \in \{0,1\}^n$ so that $x \preceq y$ and $f(y) \neq b = C(x)$.

Consider the node connecting to the output node in C. If it is an input node, say the i^{th} input node, then $f(y) \neq x_i = C(x)$. However, as $y \in \{0,1\}^n$, we also have that $y_i = C(y) = f(y)$, so $y_i \neq x_i$. As also $x \preceq y$, this entails that $x_i = \mathrm{M}$, yielding the contradiction that $b = C(x) = x_i = \mathrm{M}$.

Now consider the case that the node connecting to the output node in C is a gate. Consider the subcircuits $C_1, \dots, C_k$ computing the inputs to the gate and denote by g the gate function. We have that $g(C_1(y), \dots, C_k(y)) = C(y) = f(y) \neq C(x) = g_{\mathrm{M}}(C_1(x), \dots, C_k(x))$, where again we used that $C(y) = f(y)$ because $y \in \{0,1\}^n$. As $g_{\mathrm{M}}(x) \neq \mathrm{M}$ would entail that $g(C_1(z), \dots, C_k(z)) = g_{\mathrm{M}}(C_1(x), \dots, C_k(x))$ for all $x \preceq z \in \{0,1\}^n$, we again arrive at the contradiction that $C(x) = \mathrm{M} \neq b$. □

This theorem shows that we cannot do better than computing the metastable closure. We now show that the closure can also be implemented, using a generalized CMUX as key ingredient.

Lemma 6.5. *Let* MUX: $\{0,1\}^{2^k} \times \{0,1\}^k \to \{0,1\}$ *be a the generalized* MUX *function, i.e.,* $\mathrm{MUX}(x,s) = x_s$*, where* $s \in \{0,1\}^k$ *is interpreted as (the binary encoding of) an index. It holds that*

$$\mathrm{MUX}_{\mathrm{M}}(x,s) = b \in \{0,1\} \Leftrightarrow \forall s \preceq s' \in \{0,1\}^k \colon x_{s'} = b$$

and there is a circuit of size $\mathcal{O}(2^k)$ implementing $\mathrm{MUX}_{\mathrm{M}}$.

Proof. Exercise. □

Theorem 6.6. *For any $f \colon \{0,1\}^n \to \{0,1\}^m$, a circuit implementing f_{M} exists.*

Proof. W.l.o.g., assume that $m = 1$ (otherwise, perform the construction for each output bit of f separately). Let $f \colon y \mapsto f(y)$. By Lemma 6.5, we can implement $\mathrm{MUX}_{\mathrm{M}}$. Take such a circuit for $k = n$ and feed it inputs $x_s = f(s)$ and $s = y$, see Figure 6.4. If $f(z) = b \in \{0,1\}$ for all $y \preceq z \in \{0,1\}^n$, then by Lemma 6.5 the resulting circuit C outputs b. Thus, $f_{\mathrm{M}}(y) \preceq C(y)$ for all $y \in \{0,1,\mathrm{M}\}^n$, i.e., C implements f_{M}. □

Remarks:

- By Theorem 6.4, the circuit C from Theorem 6.6 satisfies that $C = f_{\mathrm{M}}$.
- The construction has, unfortunately, exponential size in n. Can we do better?

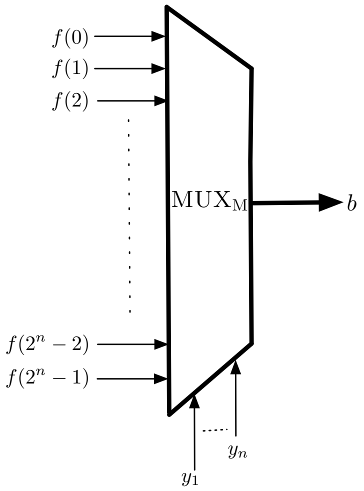


Figure 6.4: We first implement the circuit $\mathrm{MUX_M}\colon \{0,1\}^{2^n} \times \{0,1,\mathrm{M}\}^n \to \{0,1,\mathrm{M}\}$ according to the Lemma 6.5. Then we set the input domain to be $\{f(0),\ldots,f(2^n-1)\} \times \{0,1,\mathrm{M}\}^n$ as depicted in the figure.

6.3 Hardness of Containment

Recall that for any language in NP and word x, if x is in the language, there is a polynomially checkable proof w that this is the case. On the other hand, if x is not in the language, no proof w will work. If we had a small circuit implementing the metastable closure of the checker, we could exploit this to determine membership in the language efficiently.

Theorem 6.7. *Let $V_n(x,w)$, $n \in \mathbb{N}$, denote the family of verifier functions for 3SAT with n clauses, i.e., x encodes a 3SAT instance with n clauses, $w \in \{0,1\}^n$ is an assignment of these variables, and $V_n(x,w) = 1$ if and only if the instance is satisfied with the assignment given by w. If there is a family of circuits C_n, $n \in \mathbb{N}$, of size $n^{\mathcal{O}(1)}$ such that C_n implements $(V_n)_{\mathrm{M}}$, then there is a family of circuits of size $n^{\mathcal{O}(1)}$ deciding 3SAT instances on n clauses.*

Proof. We construct a circuit simulating $(V_n)_{\mathrm{M}}$. That is, we encode 0, 1, and M using two bits, e.g., 00 for 0, 11 for 1, and 01 for M. Then we construct (constant-size) circuits implementing the closure of basic gates (i.e., $\textsc{Or}_{\mathrm{M}}$, $\textsc{And}_{\mathrm{M}}$, and $\textsc{Not}_{\mathrm{M}}$) in this encoding and replace all gates in the circuit implementing $(V_n)_{\mathrm{M}}$ accordingly.

Now we use the simulating circuit as follows. For any instance x, compute $(V_n)_{\mathrm{M}}(x,\mathrm{M}^{3n})$ ($3n$ is the maximum number of variables in n clauses). If $(V_n)_{\mathrm{M}}(x,\mathrm{M}^{3n}) = 0$, output 0, otherwise output 1. We claim that this circuit, which is of size $n^{\mathcal{O}(1)}$, decides 3SAT with n clauses. To see this, assume that x is a "no" instance first. Then for any assignment w, it holds that $V_n(x,w) = 0$, implying that $(V_n)_{\mathrm{M}}(x,w) = 0$ and the output is correct. On the other hand, if x is a "yes" instance, there must be at least one witness w so that $V_n(x,w) = 1$. As $\mathrm{M}^{3n} \preceq w'$ for any $w' \in \{0,1\}^{3n}$, in particular $\mathrm{M}^{3n} \preceq w$. Accordingly, $(V_n)_{\mathrm{M}}(x,w) \neq 0$ and the output of the circuit is also correct. □

Remarks:

- The same argument applies to any problem in NP, implying that any verifier function of an NP-complete problem is unlikely to admit a small metastability-containing implementation.
- In the simulation, it is straightforward to properly "compute" with M. We're not actually providing bad inputs, we're only simulating the behavior of the containing circuits in our worst-case model, which is deterministic!
- One can even show *unconditional* exponential separations between the minimum size of non-containing and containing circuits for some explicit functions!
- So, should we accept our fate and give up? No, we still got some tricks up our sleeves! In many settings it's way to pessimistic that arbitrarily many metastable inputs can appear. We'll find small circuits that have output $f_{\mathrm{M}}(x)$ for inputs x with few Ms.

6.4 Containing a Bounded Number of Metastable Inputs

As the base case of our construction, we construct circuits handling only fixed positions for the (up to) k unstable bits. We take 2^k copies of a circuit computing f. For the i^{th} copy, we fix the k considered bits to the binary representation of i. Now we use a CMUX to select one of these 2^k outputs, where the original k input bits that we replaced are used as the select bits.

Lemma 6.8. *Let C be a circuit implementing $f : \{0,1\}^n \to \{0,1\}$ and $S \subseteq [n]$ with $|S| = k$. Denote by $|C|$ the size (i.e., number of gates) of $|C|$. Then there is a circuit of size at most $2^k(|C| + \mathcal{O}(1))$ that computes $f_{\mathrm{M}}(x)$ for any $x \in \{0, 1, \mathrm{M}\}$ satisfying that $x_i = \mathrm{M} \Rightarrow i \in S$.*

Proof. For every assignment $a \in \{0,1\}^{|S|}$ of stable values to the indices of x that are in S, compute $g_a = f(x|_{S\leftarrow a})$, where $x|_{S\leftarrow a}$ is the bit string obtained by replacing in x the bits at the positions S by the bits of vector a. We feed the results and the actual input bits from indices in S into the the k-bit $\mathrm{MUX}_{\mathrm{M}}$ given by Lemma 6.5, such that for stable values the correct output is determined. The correctness of the construction is now immediate from the properties of $\mathrm{MUX}_{\mathrm{M}}$. Concerning the size bound, for each $a \in \{0,1\}^{|S|}$ we can use C with some fixed inputs to compute g_a. Using the size bound for the MUX from Lemma 6.5, the construction thus has size $2^k(|C| + \mathcal{O}(1))$. □

Using this construction as the base case, we increase the number of sets (i.e., possible positions of the k unstable bits) our circuits can handle.

Theorem 6.9. *Let C be a circuit implementing $f : \{0,1\}^n \to \{0,1\}$. There is a circuit of size at most $(ne/k)^{2k}(|C| + \mathcal{O}(1))$ that computes $f_{\mathrm{M}}(x)$ for any $x \in \{0, 1, \mathrm{M}\}$ satisfying that $|S| \leq k$ for $S := \{i \in \{1, \ldots, n\} \mid x_i = \mathrm{M}\}$.*

Proof. Put an ordering on all k-bit subsets of $\{1, \dots, n\}$ and let S_i, $i \in \{1, \dots, I\}$ be the i^{th} element. Denote by C_{ij}, $1 \leq i, j \leq I$, a circuit whose outputs coincide with f_{M} whenever all unstable bits are from $S_i \cup S_j$. Set $a_i := \textsc{And}_{\mathrm{M}}(C_{i1}, \dots, C_{iI})$ ($\textsc{And}_{\mathrm{M}}$ with fan-in I is implemented by a binary tree of fan-in 2 $\textsc{And}_{\mathrm{M}}$ gates of minimum depth). We claim that $o := \textsc{Or}_{\mathrm{M}}(a_1, \dots, a_I)$ (implemented by a tree of fan-in 2 $\textsc{Or}_{\mathrm{M}}$ gates) coincides with f_{M} whenever there are at most k unstable bits.

To show the claim, assume that $x \in \{0, 1, \mathrm{M}\}^n$ is stable except at indices from some $S_i \in \binom{[n]}{k}$. Assume first that $f_{\mathrm{M}}(x) = 1$. Thus, we get that $a_i = \textsc{And}_{\mathrm{M}}(1, \dots, 1) = 1$. This implies $o = 1$, because the I-bit $\textsc{Or}_{\mathrm{M}}$ has a stable 1 at one of its inputs. Next, suppose that $f_{\mathrm{M}}(x) = 0$. Then, for each $1 \leq i' \leq I$, $C_{i'i}(x) = 0$. Hence $a_{i'} = 0$, because the I-bit $\textsc{And}_{\mathrm{M}}$ has a stable 0 at one of its inputs. It follows that $o = \textsc{Or}_{\mathrm{M}}(0, \dots, 0) = 0$. The case that $f_{\mathrm{M}}(x) = \mathrm{M}$ is trivial; hence the claim holds.

The above circuit contains the circuits C_{ij} and additionally $I^2 - 1$ many gates (a binary tree of $\textsc{And}_{\mathrm{M}}$ and $\textsc{Or}_{\mathrm{M}}$ gates). By Lemma 6.8, each C_{ij} can be implemented with size $2^{2k}(|C| + \mathcal{O}(1))$, as $|S_i \cup S_j| \leq 2k$. Moreover, using exactly all subsets of size $2k$, we use at most $\binom{n}{2k} \leq (en/2k)^{2k}$ different such circuits. This results in at most

$$\left(\frac{en}{k}\right)^{2k} (|C| + \mathcal{O}(1)) + \binom{n}{k}^2 - 1 = \left(\frac{en}{k}\right)^{2k} (|C| + \mathcal{O}(1))$$

gates. □

Bibliographic Notes

In the context of switching networks, hazards — changing inputs to circuits changing the output, even though the stable values are the same regardless of the changing bits — were studied even before modern computers existed [Got49], in the context of relay networks. This early Japanese work remained unnoticed in the western world, and Huffman studied the same issue, also developing a CMUX [Huf57]. Huffman noted that his design principle is sufficiently general to construct hazard-free circuits for any Boolean function. Yoeli and Rinon formalized the connection to Kleene logic [YR64] (which Goto also had done before!) or, more precisely, Kleene's strong logic of indeterminancy K_3 [Kle52, §64]. Our worst-case model for metastability (propagation) presented here results in the same logic, i.e., hazard-free circuits are the same as metastability-containing circuits. In [FFL18], a more general model for clocked circuits is presented. However, so long as only standard registers are used, the computational power is the same as that of the combinational circuits introduced in this lecture. This changes when employing masking registers, which "mask" internal metastability to the outside world by outputting a stable value; the result is that metastability may result in late transitions only, which in the worst case may result in M being read from the register in a single round.

The fact that metastability cannot be avoided in general is, in essence, a topological statement: As the output of a bistable element like a flip-flop is, due to physics, a continuous function of its input, the fact that there are two distinct stable states necessitates at least one unstable third equilibrium state. This was shown by Marino **??**. This impossibility holds also in the abstract model given here — no circuit can reliably detect or resolve metastability of its inputs [FFL18].

All of these works leave aside the complexity question, namely how large containing circuits must be. The lower bound given here is very simple, but to the best of our knowledge was first formalized by Ikenmeyer et al. [IKL+18], who also show unconditional exponential separations between containing and standard circuits based on monotone circuit complexity. The same work also gives a construction for circuits containing k bits, which is slightly weaker than the one given here. A number of works provides small circuits whose output coincides with f_{M} for specific f and certain inputs. Most notably, this is the case for sorting [BLM18], which we will study in the next lecture.

Bibliography

[BLM18] Johannes Bund, Christoph Lenzen, and Moti Medina. Optimal Metastability-Containing Sorting Networks. In *Design, Automation and Test in Europe (DATE)*, 2018. To appear. Preliminary version available at https://arxiv.org/abs/1801.07549.

[FFL18] Stephan Friedrichs, Matthias Függer, and Christoph Lenzen. Metastability-Containing Circuits. *IEEE Transactions on Computers*, 2018. To appear, online first.

[Got49] M. Goto. Application of Logical Mathematics to the Theory of Relay Networks (in Japanese). *J. Inst. Elec. Eng. of Japan*, 64(726):125–130, 1949.

[Huf57] David A. Huffman. The Design and Use of Hazard-Free Switching Networks. *J. ACM*, 4(1):47–62, 1957.

[IKL+18] Christian Ikenmeyer, Balagopal Komarath, Christoph Lenzen, Vladimir Lysikov, Andrey Mokhov, and Karteek Sreenivasaiah. On the complexity of hazard-free circuits. In *Symposium on the Theory of Computing (STOC)*, 2018. To appear. Preprint available on arxiv: https://arxiv.org/abs/1711.01904.

[Kle52] Stephen Cole Kleene. *Introduction to Metamathematics.* North Holland, 1952.

[YR64] Michael Yoeli and Shlomo Rinon. Application of Ternary Algebra to the Study of Static Hazards. *J. ACM*, 11(1):84–97, 1964.

Lecture 7

Metastability-Containing Control Loops

Like any clock synchronization algorithm (and many other distributed algorithms), one may view the Lynch-Welch algorithm as a (distributed) *control loop.* Basically, a control loop is seeking to adjust some (measurable) variable. To this end, it repeatedly or continually takes measurements and applies according adjustments, which naturally implies a mechanism to influence the variable of interest (see Figure 7.1). As measurements and corrections may be inaccurate, and the variable is also subject to influence by some external factors, the control loop must react sufficiently quickly and accurately to maintain a desired state against such unwanted "disturbances."

More concretely, for clock synchronization, the variable is the vector of correct nodes' clock values, the regulation is performed by adjusting the clocks, the external influence is given by drifting clocks, and clock drifts and uncertainty in message delays makes measurements of clock differences inaccurate. Two important aspects of control loops is whether they are operating on a continuous or discrete variable and whether the control is applied continuously or in time-discrete steps. An example for the answer being continuous in both cases is the gradient clock synchronization algorithm: logical clocks are continuous functions, and the GCS algorithm adjusts their rates. In contrast, pulse synchronization algorithms are an example for continuous variables (pulses can occur at any real time), but discrete time steps (for each $i \in \mathbb{N}$, each correct node generates exactly one pulse event).

Remarks:

- Note that the discretization is, of course, an abstraction in itself. It is implemented in a physical — and thus, neglecting quantum mechanics, continuous — world.
- If algorithms perform complicated message exchanges and computations, seeing them as control loops is usually not useful. However, the Srikanth-Toueg and Lynch-Welch algorithms can be readily interpreted as distributed control loops.
- The corrections are not applied instantaneously. It takes time to take measurements, compute a correction, and apply it. This contributes to

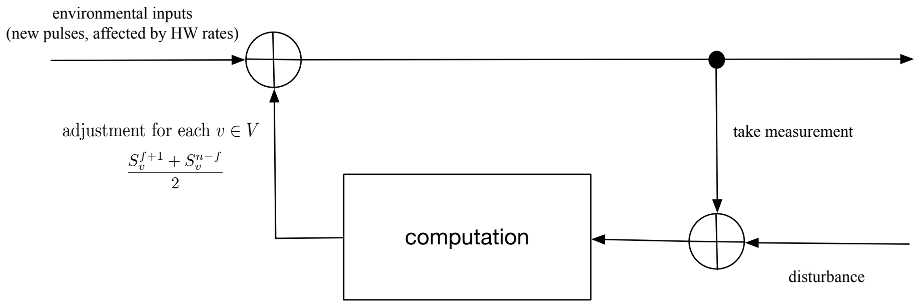

environmental inputs
(new pulses, affected by HW rates)
adjustment for each $v \in V$
$\frac{S_v^{f+1} + S_v^{n-f}}{2}$
take measurement
computation
disturbance

Figure 7.1: The whole network as a control loop.

the quality of control; in extreme cases, the control loop fails to produce anything close to the ideal behavior of the system.

- A lot of theory on control loops assumes very simple feedback mechanisms, like adjustments that are linear in the measured difference to the desired state of the system. This is not the case for our algorithms: the necessity to limit the influence of Byzantine nodes results in non-linear responses to the measurements in both algorithms.
- So why are we talking about control loops if we can't use the existing theory? In part to explain the lecture's title, and in part to clarify where metastability-containing circuits come into play.

7.1 Metastability in Control Loops

In the Lynch-Welch algorithm, we adjust continuous variables (when to generate pulses) for each round of the algorithm. The abstraction of rounds simplifies matters for us. Even better, each node in the system does this independently from the others, in the sense that we can interpret the algorithm at node $v \in V_g$ as a control loop in which all the other nodes are simply part of the environment, see Figure 7.2. But how do we actually decide how to adjust he clocks? After all, computers cannot *actually* use real values in computations. There are, essentially, two solutions:

1. Use an *analog* computation, in which all (including intermediate) values are represented by continuously-valued physical variables, like the charge of a capacitor or the amount of water in a bucket, and operate on them using continuous (physical) operations.
2. Take discrete measurements, which is done by *time-to-digital converters* (TDCs). Considering the rounding error as additional contribution to δ, one then can compute a corresponding adjustment to when the next pulse occurs, just as in the analog case.

Both approaches have their pros and cons. Analog solutions typically require specialized components, can be bulky, and require more work for adapting them to different technologies. However, they can avoid metastability altogether, as they never try to map values from a continuous to a discrete range. On the other hand, using synchronizers (i.e., time), it is straightforward to resolve metastability sufficiently reliably.

So, why not always go for the simpler, second option? The problem is that time is critical in many control loops. Recall that the Lynch-Welch algorithm guarantees a skew of $\mathcal{O}(u + (1 - 1/\vartheta)T)$, where T is the (nominal) duration of a round. We can choose $T \in \mathcal{O}(d)$, but d includes not only communication delays, but also computation. Thus, if we spend T_s time on synchronization, this adds $(\vartheta - 1)T_s$ to the skew. On a chip, it may very well be the case that T_s becomes the larger part of T, resulting in $(\vartheta - 1)T_s$ being the dominant contribution to the skew (unless local clocks are good enough). Hence, our goal for today is to remove the synchronization delay, despite sticking with the second approach!

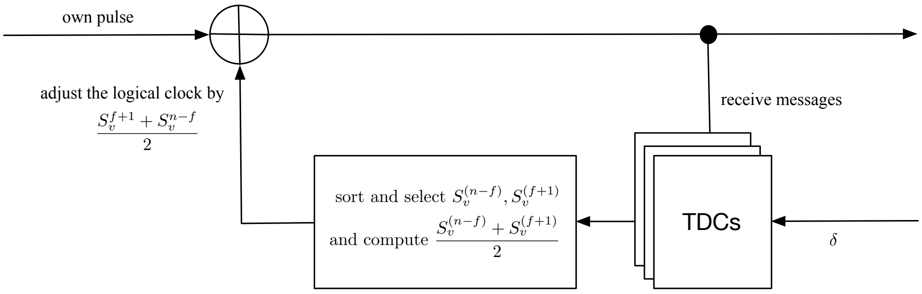

own pulse
adjust the logical clock by $\frac{S_v^{f+1} + S_v^{n-f}}{2}$
receive messages
sort and select $S_v^{(n-f)}, S_v^{(f+1)}$
and compute $\frac{S_v^{(n-f)} + S_v^{(f+1)}}{2}$
TDCs
δ

Figure 7.2: The system from the point of view of a single node — also a control loop.

7.2 First Try: Binary Counters

We need to break down the measurements and computations performed by a node executing the Lynch-Welch algorithm and implement each steps in a way that keeps (potential) metastability in check (see Figure 7.3). At each $v \in V_g$, in each round we need to

1. Send a message to each other node ϑS time after the (local) start of the round.

2. Receive the other nodes' messages and derive measurements of the difference in local time, resulting in the (unordered multi)set S_v.

3. Determine $S_v^{(f+1)}$ and $S_v^{(n-f)}$.

4. Adjust v's local clock by $(S_v^{(f+1)} + S_v^{(n-f)})/2$.

The first task is a no-brainer; we simply send the respective message $\vartheta\mathcal{S}$ local time after the time t when $L_v(t) \bmod T = \mathcal{S}$. The analysis shows that this time is unique (so the messages are indeed sent only once) and this does not require to keep track of unbounded clock values, which would be an annoying problem due to our machines having only finite memory.

The second task requires some more thought. Again, we do not want to keep track of unbounded values. Recall that Lemma 5.9 asks us to set

$$\Delta_w^v := L_v(t) - (r-1)T - (\vartheta^2 - 1)\mathcal{S} - \vartheta d\,,$$

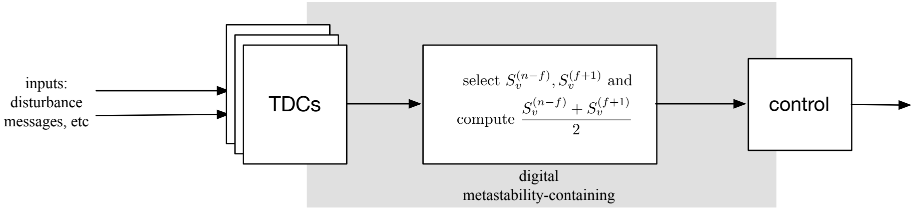

inputs:
disturbance
messages, etc
TDCs
select $S_v^{(n-f)}, S_v^{(f+1)}$ and
compute $\frac{S_v^{(n-f)} + S_v^{(f+1)}}{2}$
control
digital
metastability-containing

Figure 7.3: Control flow of a single node. The gray area uses digital logic and needs to contain metastability.

where t is the time when $v \in V_g$ receives the message for round r from $w \in V_g$. We also saw that all rounds are executed correctly (assuming we can implement the correct behavior of the nodes!), i.e., $t \in [p_{v,r}, \tau_{v,r}]$. This is good news, as we know that $L_v(p_{v,r}) = (r-1)T + \mathcal{S}$ and $L_v(\tau_{v,r}) = (r-1)T + (\vartheta^2 + \vartheta + 1)\mathcal{S} + \vartheta d$. Thus, we can simply start a counter at time $p_{v,r}$ and stop it at time t (when the message is received), where we know that the maximum (local) time difference which we the counter must be able to represent is $(\vartheta^2 + \vartheta)\mathcal{S} + \vartheta d$. Here, the counter is driven by the local clock and stopped by the arriving message. Thus, if the counter value at time t is c and the local time between consecutive up-counts of the counter is g, we have that

$$\begin{aligned} L_v(t) &\in [L_v(p_{v,r}) + cg, L_v(p_{v,r}) + (c+1)g] \\ &= [(r-1)T + \mathcal{S} + cg, (r-1)T + \mathcal{S} + (c+1)g]\,. \end{aligned}$$

Corollary 7.1. *Let $v \in V_g$ start a counter driven by its local clock at time $p_{v,r}$ that is stopped when receiving a message from node $w \in V_g$. If round r of the Lynch-Welch algorithm is executed correctly and the local time between up-counts of the counter is g, setting*

$$\Delta_v^w := cg - \vartheta^2 \mathcal{S} - \vartheta d$$

yields an estimate satisfying $\delta \leq u + (\vartheta - 1)d + 2(\vartheta^2 - \vartheta)\mathcal{S} + g$. Moreover, $c \leq ((\vartheta^2 + \vartheta)\mathcal{S} + \vartheta d)/g$.

Pretty straightforward, so all we need now is a fast counter, i.e., one for which g is sufficiently small to not matter much, right? The answer to that is an emphatic ***no!*** We have neglected that there is no guaranteed timing relation between the counter's up-counts and when the arrival of the message from w stops the counter. Here is a simple argument why this must potentially argue in metastability.

Lemma 7.2. *Assume that a counter is driven by a free-running clock source, started at time 0, and stopped at an arbitrary time $\tau \in (0, t_{\max}]$ (where $t_{\max} \geq g$ and the counter increments every g time). Let $s(\tau, t)$ be the $k \in \mathbb{N}$ bits stored in the counter's registers at an time $t > t_{\max}$ for a given τ. If this state is a continuous function of τ (w.r.t. to the standard topologies on $\mathbb{R}$ and $\{0,1\}^k$), then we cannot have that $s(\tau, t) \in \{0,1\}^k$ for all τ.*

Proof. Assume for contradiction that for any τ, $s(\tau, t) \in \{0,1\}^k$. As $t_{\max} \geq g$, this implies that there are choices $0 \leq \ell_0 \neq r_0 \leq t_{\max}$ so that $s(\ell_0, t) \neq s(r_0, t)$.

Now we apply the technique of nested intervals. For $i \in \mathbb{N}$, set $\tau := (\ell_{i-1} + r_{i-1})/2$. Clearly, $s(\tau, t) \neq s(\ell_{i-1}, t)$ or $s(\tau, t) \neq s(r_{i-1}, t)$. In the former case, set $\ell_i := \ell_{i-1}$ and $r_i := \tau$, otherwise $\ell_i := \tau$ and $r_i := r_{i-1}$. We have that

- The sequence $(\ell_i)_{i\in\mathbb{N}}$ is increasing and upper bounded by r_i for any $i \in \mathbb{N}$, hence it converges to some value $\ell^* \leq \inf_{i\in\mathbb{N}}\{r_i\}$.
- The sequence $(r_i)_{i\in\mathbb{N}}$ is decreasing and lower bounded by ℓ_i for any $i \in \mathbb{N}$, hence it converges to some value $r^* \geq \sup_{i\in\mathbb{N}}\{\ell_i\}$.
- We have that $\ell^* = r^*$, as $\lim_{i\to\infty}(r_i - \ell_i) = 0$.
- By continuity of $s(\cdot, t)$, we have that $s(\ell^*, t) = \lim_{i\to\infty} s(\ell_i, t)$. As $\{0,1\}^k$ is a discrete space, this means that there is some $i_\ell \in \mathbb{N}$ so that $s(\ell_i, t) = s(\ell^*, t)$ for all $i \geq i_\ell$.
- Likewise, there is some i_r so that $s(r_i, t) = s(r^*, t)$ for all $i \geq i_r$.
- We have that $s(\ell_i, t) \neq s(r_i, t)$ for all $i \in \mathbb{N}_0$ by construction.

Altogether, we arrive at the contradiction that, for any $i \geq \max\{i_\ell, i_r\}$, it holds that $s(\ell_i, t) \neq s(r_i, t) = s(r^*, t) = s(\ell^*, t) = s(\ell_i, t)$. □

Remarks:

- If you are puzzled by the lemma requiring the "standard topologies," don't worry about it. On $\mathbb{R}$, this simply means the open and closed sets you know. On $\{0,1\}^k$, just intersect the open and closed sets in $\mathbb{R}^k$ with $\{0,1\}^k$ to get the open and closed sets, respectively. As a ball of radius smaller than 1 around a point in $\{0,1\}^k$ just contains the point, this means that any convergent series becomes constant at some point. This is what we used in the proof.
- These choices of topologies actually make sense. Any physical circuit will respond to continuous changes of its input with continuous changes of the output. However, we want stable and clearly distinguishable values in our registers. This means to consider clearly separated regions of the state space: The "0-region" of a register's (physical) state space should be clearly separated from its "1-region." This separation means that a small change cannot make the register "jump" from the 0- to the 1-region— which is reflected by the discrete topology on $\{0,1\}$.
- By inserting M as a third value covering the "gap" between 0 and 1, we can properly reflect that circuits cannot do this job. In the topology, this is reflected by the fact that no matter how small a ball becomes, it doesn't separate the 0- and the M-region of the register's state space. We defined that M stands for *any* state that is not in the 0- or the 1-region!

Does this mean we're in trouble? In the previous lecture we saw that we can deal with metastability to some extent. Unfortunately, following conventional wisdom won't work here.

Corollary 7.3. *Consider the same setting as in Lemma 7.2. If the counter uses standard binary encoding and $t_{\max}$ is large enough for it to count up to 2^b, $b \in \mathbb{N}_0$, then we can force the counter register holding the $(b+1)$-least significant bit to be* M *at any time $t > t_{\max}$.*

Proof. We use essentially the same argument, but we start from more specific times ℓ_0 and r_0. As the counter can count up to 2^b, we can choose ℓ_0 and r_0 such that $s(\ell_0, t) = 0 \dots 01 \dots 1$ and $s(r_0, t) = 0 \dots 010 \dots 0$, where we wrote the least significant bits to the right and in each case the identical bits to the right are k many. This follows from the fact that the counter increment from 2^{b-1} to 2^b must change the register states between these two (stable) states. Now we can construct our nested intervals by performing our case distinction according to the $(k+1)^{th}$ bit (counting from the least significant one). By the same arguments as before, we obtain a time at which the bit cannot be stable and therefore must be M. □

Remarks:

- Unless one is very careful when implementing the counter, things actually get worse: we may end up with state $0 \dots 0\mathrm{M} \dots \mathrm{M}$. In case the full range of the counter is utilized, we may face a memory state of $\mathrm{M} \dots \mathrm{M}$!
- Think about this for a second. We started with being uncertain whether an up-count of the counter took place or not, because the counter was stopped in the middle of an increment. But we lost *all* information about the relative timing of the start of the counter and the stop signal!
- Even if the counter was particularly cleverly implemented, Corollary 7.3 shows that we might end up with very wrong encoded values.
- The problem here lies with the encoding. When containing metastability, the encoding matters!

7.3 Second Try: Unary "Counters"

We need to look for an encoding without such flaws. A very simple solution is to use a unary encoding. In a B-bit unary code, $k \in [B+1]$ is represented by $1^k 0^{B-k}$. A unary code "counter" is implemented by a *delay line,* which consists of a sequence of B buffers of uniform delay g, where we connect to each stage the set input of a register (which is initialized to 0). The counter is stopped by latching all registers on occurence of the stop signal. When g is large enough to guarantee that only a single register is unstable (transitioning) at any given point, at most one register ends up in a metastable state when stopping the counter. This is a sensible measure in most cases, as otherwise even after stabilization we may end up with a stored string like 11101000. Basically, we can't make the measurement more accurate than imposed by the speed at which registers can be set.

Alright, we measured time differences in terms of unary encodings, $v \in V_g$ has for each $w \in V_g$ stored a time difference in unary, i.e., a B-bit string of the form $1 \dots 10 \dots 0$ or, possibly, $1 \dots 1\mathrm{M}0 \dots 0$. We refer to these strings as s_v^w, and can easily translate them to the measured time difference by multiplication with g (up to an error of up to roughly g), where M can be interpreted either way.

Our next step is to determine which of these strings represent $S_v^{(f+1)}$ and $S_v^{(n-f)}$, respectively. One way of doing this is to sort the strings. For this to be meaningful, we need to give a total order on the potential input strings. The only sensible order is in accordance with the time measured.

Definition 7.4 (Total Order of Inputs for Unary Encoding)**.** *Consider the set of strings*

$$U_B := \{1^k0^{B-k} \,|\, k \in [B+1]\} \cup \{1^k\mathrm{M}0^{B-k-1} \,|\, k \in [B]\}\,.$$

For $x, y \in U_B$,

$$x \leq_U y \Leftrightarrow \begin{cases} k \leq k' & \textit{for } x = 1^k0^{B-k} \textit{ and } y = 1^{k'}0^{B-k'} \\ k \leq k' & \textit{for } x = 1^k0^{B-k} \textit{ and } y = 1^{k'}\mathrm{M}0^{B-k'-1} \\ k \leq k' & \textit{for } x = 1^k\mathrm{M}0^{B-k-1} \textit{ and } y = 1^{k'}\mathrm{M}0^{B-k'-1} \\ k < k' & \textit{for } x = 1^k\mathrm{M}0^{B-k-1} \textit{ and } y = 1^{k'}0^{B-k'-1}. \end{cases}$$

A crucial observation is that this order is also sensible in another regard: When resolving metastability, a string does not "pass" any stable strings in the order. We can apply the results from the previous lecture to see that sorting according to this order is indeed possible with a circuit.

Lemma 7.5. *Given n strings from U_B, there is a circuit sorting them according to the order from Definition 7.4.*

Proof. We claim that the metastable closure of the function sorting the stable inputs sorts according to the order from Definition 7.4. The statement of the lemma then follows from Theorem 6.6.

To see this, sort a fixed set of input strings in accordance with the order and consider the i^{th} output string. If it is stable, observe that picking arbitrary stabilizations for the other strings and sorting accordingly will not change the string in position i, as stabilizing a string $1^k\mathrm{M}0^{B-k-1}$ does not move it "past" any stable string in the order. On the other hand, if the string is not stable, i.e., $1^k\mathrm{M}0^{B-k-1}$ for some $k \in [B]$, observe that stabilizing all input strings by replacing M with 0 results in the sorted sequence having 1^k0^{B-k} on position i in the sorted list (as nothing moves past stable strings). Likewise, stabilizing all input strings by replacing M with 1 results in the sorted sequence having $1^{k+1}0^{B-k-1}$ in position i, so bit $k+1$ of the i^{th} output must be M. Any other stabilization will result in either 1^k0^{B-k} or $1^{k+1}0^{B-k-1}$ on position i. The claim follows, completing the proof. □

However, using the construction from Theorem 6.6 would result in a circuit of exponential size, so let's be more clever. In absence of metastability, *sorting networks* are simple and fast solutions to compute what we need.

Definition 7.6 (Sorting Network)**.** *An n-input sorting network consists of n parallel* wires *oriented from left to right and a number of* comparators *(cf. Figure 7.4). Each comparator connects two of the wires, by a straight connection orthogonal to the wires. Moreover, no two of the comparators connect to the same point on a wire.*

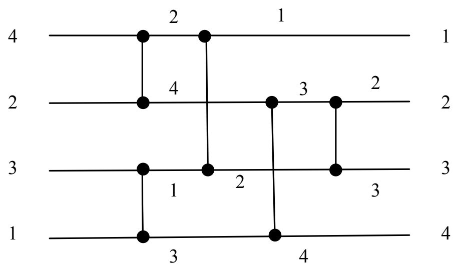


Figure 7.4: A sorting network with four inputs. Each comparator performs a compare and (if necessary) swap operation of its two inputs. The outputs are shown for the input sequence $(4, 2, 3, 1)$.

A sorting network is fed an input from a totally ordered set to the start of each wire. Each comparator takes the two inputs provided by to it, outputting the larger input to the top wire and the smaller input to the bottom wire. A correct sorting network guarantees that for any choice of inputs, the outputs are the sequence resulting from ordering the inputs descendingly from top to bottom.

Sorting networks are understood very well. Constructions that are simultaneously (asymptotically) optimal both with respect to size — the total number of comparators — and depth — the maximum number of comparators "through" which a value passes — are known. Conveniently, sorting networks are correct if and only if the correctly sort 0s and 1s, so it suffices if we can figure out how to implement a comparator that correctly sorts two values according to our chosen order.

Lemma 7.7. *A correct comparator implementation for unary encoding is given by the bit-wise* Or *for the upper and the bit-wise* And *for the lower output.*

Proof. Follows from the behavior of the basic gates and a case distinction. □

With sorting in place, we can determine $S_v^{(f+1)}$ and $S_v^{(n-f)}$; refer to the encodings of these values as $s_v^{(f+1)}$ and $s_v^{(n-f)}$, respectively. It remains to perform the last step, the phase correction. One solution would be an analog control of the oscillator that serves as the local clock of v. Unfortunately, such an approach is too slow or too inaccurate in practice; either would defeat the purpose of our approach. A fast "digital" solution is to have the local clock drive a counter that basically counts modulo T (where T is represented as a multiple of the time for a counter increment) and adjust this counter. Unfortunately, this is unsafe when the adjustment values suffer from potential metastability: The counter registers could become metastable, causing all kinds of problems.

Of course, we could wait for stabilization first and *then* apply the corrections to such a counter. But in that case we wouldn't have to jump through all these hoops in the first place — if we're not able to apply the computed phase correction right away, we could have waited for stabilization *before* computing it, without losing time and saving us a lot of trouble. There is something else we can do, however. We can use the unary encoded values in a delay line to shift the clock in a safe way despite metastability. Of course, we cannot have a delay line for each round of the Lynch-Welch algorithm (that would be infinite memory again!), but we can use a few in a round-robin fashion. The one which was written the longest time ago then has stabilized with sufficiently large probability to risk transferring the respective phase shift into our counter — while being used to shift the clock, the registers of the delay line have simultaneously operated as a synchronizer! See Figure 7.5 for an overview of the circuit.

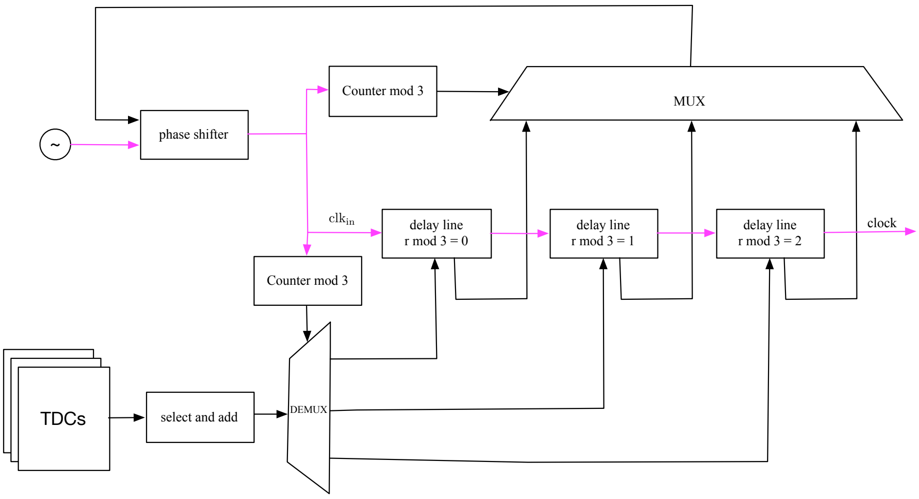


Figure 7.5: Rough overview of a circuit using a (non-containing) phase shifter and several delay lines to perform the phase shifts required by the Lynch-Welch algorithm. The delay lines are used in a round-robin fashion. In between two consecutive clock pulses, the current value held by the delay line which is to be rewritten yet is provided to the phase shifter as input, it adjusts its internal counter accordingly (making the phase shift permanent), and the registers of the delay element are latched to the current output of the computational logic. All this needs to be performed in the right order and be complete before the next pulse propragates through the phase shifter and the delay lines; the complete design requires additional circuitry ensuring this and a corresponding timing analysis.

There's still a catch: As we may have a metastable register in the delay line, the respective And gate will output a bad signal when the clock flank arrives. This would be remedied shortly after, when the delayed clock signal reaches the next stage (with a stable 0 in the register), as then the Or will have a stable input. The solution is a *high-threshold* inverter, which switches from output 1 to output 0 at a higher voltage threshold, thus "masking" the bad medium voltage. Figure 7.6 shows how the resulting delay lines look like.

Remarks:

- This works, but is still inefficient. Unary encodings are exponentially larger than binary encodings!

- Let's do better, using an encoding without redundancy that also changes only a single bit on each up-count!

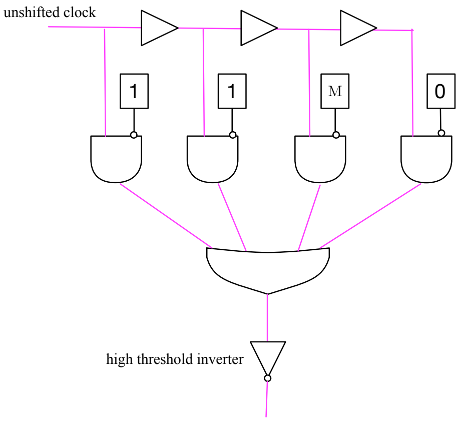


Figure 7.6: Straightforward delay line implementation. The high-threshold inverter at the output ensures that metastability is "masked," effectively transforming it into a (potentially) late, but clean transition. As a metastable register may stabilize at any time (and to either value), this may result in any delay between what we would get for a stable 0 or 1 in the register, respectively.

7.4 Third Try: Gray Codes

Unary encoding worked, but results in large circuits. A B-bit unary encoding can represent only $B+1$ different values, while a binary encoding has 2^B codewords. Binary encoding causes trouble, because a bit that may become metastable *due to an interrupted up-count* makes a huge difference with respect to the encoded value. We need a code where each up-count changes exactly one bit.

Definition 7.8 (Gray Code)**.** *A B-bit Gray code $G\colon [2^B] \to \{0,1\}^B$ maps its range $[2^B]$ one-to-one to $\{0,1\}^B$, with the property that for $x, x+1 \in [2^B]$, the resulting codewords differ in a single bit.*

Transforming unary encoding to Gray code is easy, even in face of metastability. However, we need some notation.

Definition 7.9. *For $x, y \in \{0,1,\mathrm{M}\}^k$, $k \in \mathbb{N}$, set*

$$(x * y)_i := \begin{cases} 1 & \text{if } x_i = y_i = 1 \\ 0 & \text{if } x_i = y_i = 0 \\ \mathrm{M} & \text{else.} \end{cases}$$

It is easy to see that $x * y$ is the largest common predecessor of x and y with respect to $\preceq$, i.e., $x * y \preceq x$, $x * y \preceq y$, and if $z \in \{0,1,\mathrm{M}\}^k$ satisfies $z \preceq x$ and $z \preceq y$, then $z \preceq x * y$. In other words, $x * y$ is the "most stable" string so that both x and y are stabilizations of it.

Lemma 7.10. *Let $s \in U_{2^B-1}$ for some $B \in \mathbb{N}$. If $s \in U_{2^B-1} \cap \{0,1\}^{2^B-1}$, denote the encoded number by $x \in [2^B]$. For $s \in U_{2^B-1} \setminus \{0,1\}^{2^B-1}$, let $x, x+1 \in [2^B]$*

denote the numbers encoded by the stabilizations of s. For any fixed Gray code G, there is a circuit of size $\mathcal{O}(2^B)$ and depth at most $\mathcal{O}(B)$ that computes

$$\begin{array}{cl} G(x) & \textit{if } s \in U_{2^B-1} \cap \{0,1\}^{2^B-1} \\ G(x) * G(x+1) & \textit{if } s \in U_{2^B-1} \setminus \{0,1\}^{2^B-1}. \end{array}$$

from input s.

Proof. For bit i of the code, consider the subset of $[2^B+1] \setminus \{0\}$, for which an up-count to the respective value changes bit i. We connect all corresponding registers by a tree of (2-input) Xor gates. Such a Xor tree implements an Xor with more inputs, i.e., it keeps track of the number of times the i^{th} bit changed. Accordingly, depending on whether the i^{th} bit is 0 or 1 in the first bit, this circuit generates the correct output or its conjugate for stable values; in the latter case, we simply add a Not gate. The i^{th} output bit can only become M if some input to the respective Xor tree is M. However, in this case, the respective output bit transitions on the up-count corresponding to the register holding the respective bit of the unary encoding, so the output bit ought to be M.

Concerning the complexity, the total number of Xor gates needed is $2^B - 1 - B$ (the number of input bits minus the number of output bits), plus up to B Not gates. By balancing the Xor trees, their depth becomes bounded by the logarithm of their size (rounded up). If no Xor gates are available, we can implement them by constant-sized subcircuits composed of basic gates, increasing size and depth of the circuit by constant factors only. □

Remarks:

- This is promising: $G(x) * G(x+1)$ has only a single metastable bit, as $G(x)$ and $G(x+1)$ differ only in a single bit.
- This means that there are exactly two stabilizations of $G(x) * G(x+1)$, namely $G(x)$ and $G(x+1)$. We did not lose information, and Theorem 6.6 shows that we can convert the Gray code back to unary, even with metastability!
- The circuit for this provided by the Theorem 6.6 will have exponential size, but this time this doesn't matter as much, as the output already has exponential size by itself! One can still do better (you will do so in one of the exercises).
- For this to pay of, we now need very efficient circuits for sorting Gray codes, including strings of the form $G(x) * G(x+1)$. Ordering $G(x) \leq G(x)*G(x+1) \leq G(x+1)$ and arguing analogously to Lemma 7.5, we know that we can design suitable comparators *in principle,* which then can be used in sorting networks. In the next lecture, we will find asymptotically optimal comparator circuits for a simple, convenient Gray code.

Bibliographic Notes

The concept of implementing Lynch-Welch using metastability-containing logic was proposed in [FFL18], where it was shown to be feasible. However, the underlying construction was generic (as in Theorem 6.6), resulting in large circuits.

Such circuits would incur computational delays negating the advantage of not requiring synchronizers. In [FKLP17], TDCs are given that can directly output Binary Reflected Gray Code (BRGC) with the same guarantees as provided by Lemma 7.10. This further reduces the depth and size of circuits for follow-up computations, as the conversion circuit can be skipped. Various comparators for such BRGC values have been proposed in [BLM18, BLM17, LM16]; we will discuss the currently best one next lecture. The idea for using the computed phase shifts in delay lines until they have stabilized with sufficient probability is, essentially, applied to a different problem in [FKLW18].

Asymptotically optimal sorting networks were given in [AKS83]. For a proof that sorting networks are correct if and only if they correctly sort 0-1 inputs, see [Knu98].

Bibliography

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $\mathcal{O}(n \log n)$ Sorting Network. In *15th Symposium on Theory of Computing (STOC)*, 1983.

[BLM17] Johannes Bund, Christoph Lenzen, and Moti Medina. Near-Optimal Metastability-Containing Sorting Networks. In *Design, Automation, and Test in Europe (DATE)*, 2017.

[BLM18] Johannes Bund, Christoph Lenzen, and Moti Medina. Optimal Metastability-Containing Sorting Networks. In *Design, Automation and Test in Europe (DATE)*, 2018. To appear. Preliminary version available at https://arxiv.org/abs/1801.07549.

[FFL18] Stephan Friedrichs, Matthias Függer, and Christoph Lenzen. Metastability-Containing Circuits. *IEEE Transactions on Computers*, 2018. To appear, online first.

[FKLP17] Matthias Függer, Attila Kinali, Christoph Lenzen, and Thomas Polzer. Metastability-Aware Memory-Efficient Time-to-Digital Converters. In *Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2017.

[FKLW18] Matthias Függer, Attila Kinali, Christoph Lenzen, and Ben Wiederhake. Fast All-Digital Clock Frequency Adaptation Circuit for Voltage Droop Tolerance. In *Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2018. To appear. Preprint available at https://people.mpi-inf.mpg.de/∼clenzen/pubs/FKLW18droop.pdf.

[Knu98] Donald E. Knuth. *The Art of Computer Programming Vol. 3: Sorting and Searching.* Addison-Wesley, 1998.

[LM16] Christoph Lenzen and Moti Medina. Efficient Metastability-Containing Gray Code 2-Sort. In *Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2016.

Lecture 8

Metastability-Containing Sorting

Last week we saw how to obtain an MC implementation of a node's logic for the Lynch-Welch algorithm. However, for this to matter, we need low-depth circuits performing the computations. Otherwise, we would lose the speed advantage gained from forgoing synchronizers, meaning that all that work is for nothing! Hence, our task today is to construct low-depth sorting networks — which, as we have seen, means to construct low-depth comparators.

Before constructing the circuits, we need to fix an encoding. We already decided that we (need) to use a Gray code, but not which one. One of the simplest, if not most natural, Gray codes turns out to be well-suited for our purposes.

Definition 8.1 (Binary Reflected Gray Code)**.** B-bit Binary Reflected Gray Code (BCRG) $G_B\colon [2^B] \to \{0,1\}^B$ *is defined recursively by*

$$\begin{aligned} G_1(0) &= 0\\ G_1(1) &= 1\\ \forall B>1\,\forall x\in[2^{B-1}]\colon G_B(x) &= 0G_{B-1}(x)\\ \forall B>1\,\forall x\in[2^{B}]\setminus[2^{B-1}]\colon G_B(x) &= 1G_{B-1}(2^B-1-x)\,. \end{aligned}$$

G_B is one-to-one, so we denote by D_B its inverse, the decoding function. In the following, we will write $D(g)$ instead of $D_B(g)$, as B can be inferred from the length of the decoded string g.

We know that we won't have to handle arbitrary metastable strings, as metastability is only introduced by a TDC up-count being interrupted.

Definition 8.2 (Valid Strings)**.** *The set* valid B-bit strings *is defined as*

$$V_B \coloneqq \{G_B(x)\,|\,x\in[2^B]\}\cup\{G_B(x)*G_B(x+1)\,|\,x\in[2^B-1]\}\,.$$

We define a total order $\leq_G$ on V_B according to the encoded values. The total order is given by the transitive closure of the partial order

$$\begin{aligned} \forall g,h\in V_B\cap\{0,1\}^B\colon g<h \Leftrightarrow D(g)<D(h)\\ \forall x\in[2^B-1]\colon G(x)<G(x)*G(x+1)<G(x+1)\,. \end{aligned}$$

0	0000	4	0110	8	1100	12	1010
0-1	000M	4-5	011M	8-9	110M	12-13	101M
1	0001	5	0111	9	1101	13	1011
1-2	00M1	5-6	01M1	9-10	11M1	13-14	10M1
2	0011	6	0101	10	1111	14	1001
2-3	001M	6-7	010M	10-11	111M	14-15	100M
3	0010	7	0100	11	1110	15	1000
3-4	0M10	7-8	M100	11-12	1M10	---	---

Table 8.1: Valid 4-bit strings.

Denote by $\max_G$ *and* $\min_G$ *the maximum and minimum w.r.t. to* $\leq_G$*.*

Table 8.1 list V_B according to $\leq_B$. Our goal is to compute $\max_G$ and $\min_G$ for given valid strings $g, h \in V_B$. As you have shown in an exercise, for inputs that are valid strings the above definitions of $\max_G$ and $\min_G$ coincides with the metastable closure of their restrictions to stable values, i.e.,

$$\max_G\{g,h\} = \mathop{*}_{\substack{g \preceq g' \in \{0,1\}^B \\ h \preceq h' \in \{0,1\}^B}} \max_G(g',h')\,.$$

Thus, we need to figure out how to implement the closure of these (restricted) operators, at least for inputs that are valid strings.

8.1 4-valued Comparison of BRGC Strings

Our first step is to break down the task of determining $\max_G$ into smaller pieces. One way of doing this is to see how a (simple) state machine can perform the

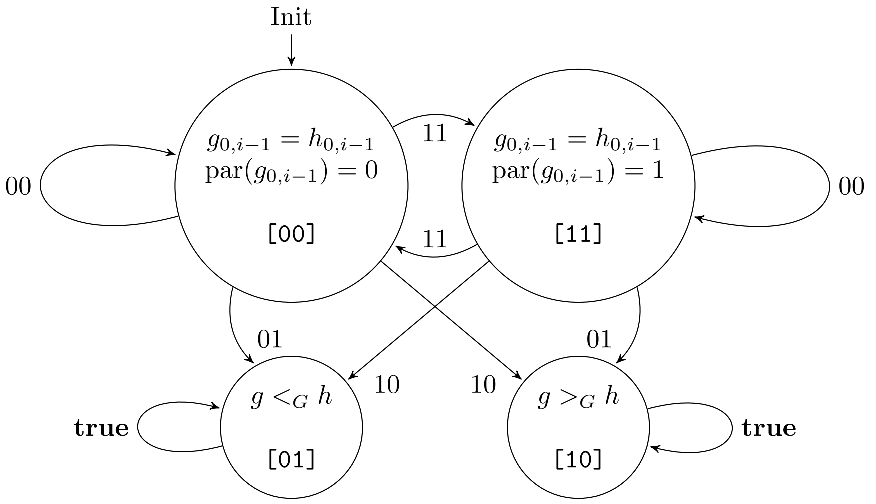


Figure 8.1: Finite state automaton determining which of two Gray code inputs $g, h \in \{0,1\}^B$ is larger. In each step, the machine receives $g_i h_i$ as input. State encoding is given in square brackets.

required computation. Our state machine is fed the input bits one pair at a time, see Figure 8.1, to determine which of the strings (if any) is larger; one then needs to determine the output accordingly. As we are dealing with Gray code, we do not have a 3-valued comparison to make (larger, smaller, or equal, non-trivially recursing only on state equal), but rather a 4-valued one: the possible states are larger, smaller, equal with even parity (standard recursion), and equal with odd parity (recurse with flipped meanings of larger and smaller).

It is straightforward to see that the state machine operates correctly on stable inputs. But what happens for unstable inputs? This is the reason why the state machine also specifies how to encode its states. We want that if at some point the state machine is not yet decided and one of the inputs is M (but the other not), the metastable closure of the state transition function yields a new "state" whose stabilizations correspond to the results of the comparisons if we had stabilized the inputs first. To formalize this, let us first fix some notation.

Definition 8.3 (Transition Operator). *Given state $s \in \{0,1\}^2$ of the state machine in Figure 8.1 and inputs $b \in \{0,1\}^2$, denote by $s \diamond b$ the resulting state of the state machine, i.e.:*

`meaning of state`	$\diamond$	`00`	`01`	`11`	`10`
`equal,` $\mathrm{par} = 0$	`00`	`00`	`01`	`11`	`10`
$<_G$	`01`	`01`	`01`	`01`	`01`
`equal,` $\mathrm{par} = 1$	`11`	`11`	`10`	`00`	`01`
$>_G$	`10`	`10`	`10`	`10`	`10`

Note that $\diamond$ is associative and $00 \diamond b = b$, so the state of the machine after processing the input completely is $\Diamond(g,h) := \Diamond_{i=1}^{B} g_i h_i := g_1 h_1 \diamond g_2 h_2 \diamond \ldots \diamond g_B h_B$, where the order in which the $\diamond$ operations are executed is arbitrary.

Lemma 8.4. *For $g, h \in \{0,1\}^B$,*

$$\Diamond(g,h) = \begin{cases} 00 & \text{if } g = h \text{ and } \mathrm{par}(g) = 0 \\ 11 & \text{if } g = h \text{ and } \mathrm{par}\, g = 1 \\ 01 & \text{if } g <_G h \\ 10 & \text{if } g >_B h\,. \end{cases}$$

Proof. We proof this by induction on B. For $B = 1$, we readily see that the state machine transitions to the correct state. For $B > 1$, observe that if the machine is in state 01 or 10 before processing the last pair of bits, by induction hypothesis $g_{1\ldots B-1} \neq h_{1\ldots B-1}$ and the machine has already decided correctly. If $g_{1\ldots B-1} = h_{1\ldots B-1}$, the parity of $g_{1\ldots B-1}$ correctly kept track of whether the remaining (trivial, 1-bit) code is listed in default order (parity 0, current state 00) or reversed (parity 1, current state 11). Checking the state transitions of the machine, we see that the machine correctly which string is larger if $g_B \neq h_B$, and correctly adjusts the parity if $g_B = h_B$. □

The state machine will have to be implemented by some circuit. From Theorem 6.6, we know that we can implement $\diamond_{\mathrm{M}}$, the metastable closure of $\diamond$. Conveniently, this operator turns out to be associative as well.

Lemma 8.5. $\diamond_{\mathrm{M}}$ *is associative.*

Proof. While an elegant proof would be desirable, all we know is how to do this by a case distinction. This is not very practical by hand (3^8 cases!), so it has been checked by machine only. □

This means that we can apply the same notation as for $\diamond$ to $\diamond_{\mathrm{M}}$ with impunity, i.e.,

$$\Diamond_{\mathrm{M}}(g,h) := \left(\Diamond_{\mathrm{M}}\right)_{i=1}^{B} g_i h_i := g_1 h_1 \diamond_{\mathrm{M}} g_2 h_2 \diamond_{\mathrm{M}} \ldots \diamond_{\mathrm{M}} g_B h_B \, .$$

In order to show that we can decompose $(\Diamond)_{\mathrm{M}}$ into repeatedly applying $\diamond_{\mathrm{M}}$ (by Lemma 8.5 in arbitrary order!), we need the following helper lemma.

Lemma 8.6. *For $g,h \in V_B$ and any $(\Diamond)_{\mathrm{M}}(g,h) \preceq s' \in \{0,1\}^2$, there are $g \preceq g' \in \{0,1\}^B$ and $h \preceq h' \in \{0,1\}^B$ such that $s' = \Diamond(g',h')$.*

Proof. If $g,h \in \{0,1\}^B$, the statement is trivially true. W.l.o.g., assume that $g \in V_B \setminus \{0,1\}^B$ (if this holds only for h, reason symmetrically). Denote by d the distance of g and h in the total order on V_B (i.e., their distance in the list given in Table 8.1). If $d > 2$, Lemma 8.4 shows that for all stabilizations of g and h, the state machine outputs the same stable value; in this case, again the statement is trivially true.

If $d = 2$ or $d = 1$, checking all possibilities (again, this is easiest using the table) and using Lemma 8.4 we see that the different stabilizations result in outpus (i) 00 and 01, (ii) 01 or 11, (iii) 11 or 10, or (iv) 10 or 00. Either way, the claim of the lemma holds: we have exactly one metastable state bit in the end, and both respective outputs can be also generated by stabilizing the inputs first.

The final case is $d = 0$, i.e., $g = h$. In this case, any output can be generated depending on how we choose to stabilize the unstable bits in g and h, respectively, and $(\Diamond)_{\mathrm{M}}(g,h) = \mathrm{MM}$. □

We can now prove the key result that implementing the metastable closure of the statemachine's transition function is sufficient to implement the closure of the comparison.

Theorem 8.7. *Let $g,h \in V_B$. Then, for any $j \in \{1,\ldots,B\}$,*

$$(\Diamond)_{\mathrm{M}}(g_{1\ldots j}, h_{1\ldots j}) = \Diamond_{\mathrm{M}}(g_{1\ldots j}, h_{1\ldots j}) \, .$$

Proof. The recursive definition of BRGC and valid strings (see Definitions 8.1 and 8.2) ensure that prefixes of valid strings are valid strings, hence it is sufficient to consider the special case that $j = B$. We prove the claim by induction on B, where the base case of $B = 1$ is trivial. For the step from $B - 1 \in \mathbb{N}$ to B, the induction hypothesis yields that

$$s := (\Diamond)_{\mathrm{M}}(g_{1\ldots B-1}, h_{1\ldots B-1}) = \Diamond_{\mathrm{M}}(g_{1\ldots B-1}, h_{1\ldots B-1}) \, .$$

By Lemma 8.6, for any $s \preceq s' \in \{0,1\}^2$, there are $g_{1\ldots B-1} \preceq g'_{1\ldots B-1} \in \{0,1\}^{B-1}$

and $h_{1\dots B-1} \preceq h'_{1\dots B-1} \in \{0,1\}^{B-1}$ so that $s' = \diamondsuit(g_{1\dots B-1}h_{1\dots B-1})$. Thus,

$$\begin{aligned}
(\diamondsuit)_{\mathrm{M}}(g,h) &= \mathop{*}_{\substack{g\preceq g'\in\{0,1\}^B\\ h\preceq h'\in\{0,1\}^B}} \diamondsuit(g'h')\\
&= \mathop{*}_{\substack{g_{1\dots B-1}\preceq g'_{1\dots B-1}\in\{0,1\}^{B-1}\\ h_{1\dots B-1}\preceq h'_{1\dots B-1}\in\{0,1\}^{B-1}\\ g_B\preceq g'_B\in\{0,1\}\\ h_B\preceq h'_B\in\{0,1\}}} (\diamondsuit(g_{1\dots B-1}h_{1\dots B-1})\diamond g_Bh_B)\\
&= \mathop{*}_{\substack{s\preceq s'\in\{0,1\}^2\\ g_B\preceq g'_B\in\{0,1\}\\ h_B\preceq h'_B\in\{0,1\}}} (s'\diamond g'_Bh'_B)\\
&= s\diamond_{\mathrm{M}} g_Bh_B\\
&= \diamondsuit_{\mathrm{M}}(g,h)\,.
\end{aligned}$$

□

Remarks:

- Lemma 8.6 can be generalized to arbitrary operators so long as only a single bit becomes metastable: In this case, we have two stabilizations of the output, meaning there must be two stabilizations of the input yielding different values.
- However, as soon as two output bits become metastable, this simple relation may break down. For instance, already making two copies of the Xor of two bits showcases this issue. Stabilizations of the inputs always result in identical output bits, but e.g. input 1M yields output MM, which may also stabilize to 01 and 10.
- We cannot always reliably decide which of the inputs to our comparator is larger, even if they are not equal. This means that computing the output is not as easy as in the binary world.

8.2 Determining the Output Bits

For stable strings, determining the output would now be straightforward. Compute $s = \diamondsuit(g,h)$, and then, e.g., pick g if $s_1 = 1$ (implying $g \geq_G h$) and h otherwise (implying $g \leq_G h$). By now, you already guess that we cannot simply use a standard MUX for this task, but need to use a MUX$_{\mathrm{M}}$. Alas, we still run into trouble with this approach.

Example 8.8. *Consider* 1*-bit inputs* $g = \mathrm{M}$ *and* $h = 1$*, i.e.,* $s = \mathrm{M}1$*. Then* $\mathrm{CMUX}(g,h,s_1) = \mathrm{CMUX}(\mathrm{M},1,\mathrm{M}) = \mathrm{M}$*, yet* $\max_G\{g,h\} = h = 1$.

One can try around, but the problem persists. The issue is that we cannot reliably decide which value is larger, so the inputs need to "help" with masking metastability. For a single bit, of course all we need to do is to feed the inputs to an Or gate. However, when looking at longer codes, the parity comes into play. So let us see what we get if we combine the state

$$s^{(i-1)} := \diamondsuit_{\mathrm{M}}(g_{1\dots i-1}, h_{1\dots i-1})$$

of the state machine *before* processing the i^{th} bits (where $s^{(0)} := 00$) with the i^{th} bits themselves to determine the i^{th} bit of the output. Taking into account the meaning of the state bits, for stable inputs this results in the following mapping $\text{out}\colon (s, g_i h_i) \mapsto \max_G\{g,h\}_i \min_G\{g,h\}_i$.

meaning of state	$s^{(i-1)}$	$\max_G\{g,h\}_i$	$\min_G\{g,h\}_i$
equal, par = 0	00	$\max\{g_i, h_i\}$	$\min\{g_i, h_i\}$
$<_G$	10	g_i	h_i
equal, par = 1	11	$\min\{g_i, h_i\}$	$\max\{g_i, h_i\}$
$>_G$	01	h_i	g_i

meaning of state	out	00	01	11	10
equal, par = 0	00	00	10	11	10
$<_G$	01	00	10	11	01
equal, par = 1	11	00	01	11	01
$>_G$	10	00	01	11	10

Note that out has 4 input bits, so the circuit implementing out_M guaranteed by Theorem 6.6 has constant size. However, this is useful only if indeed $\text{out}_\text{M}(s^{(i-1)}, g_i h_i) = \max_G\{g,h\}_i \min_G\{g,h\}_i$ all $g, h \in V_B$ and $i \in \{1, \ldots, B\}$. Proving this is simplified by the following observation on the structure of valid strings.

Observation 8.9. *If for a valid string $g \in \{0,1\}^B$ it holds that $g_i = \text{M}$ for some $i < B$, then $g_{i+1\ldots B} = 10^{B-i-1}$, i.e., $g_{i+1\ldots B}$ is the codeword for the largest value that $(B-i)$-bit Gray code can encode.*

Theorem 8.10. *Given valid inputs $g, h \in V_B$, for all $i \in \{1, \ldots, B\}$ it holds that $\text{out}_\text{M}(s^{(i-1)}, g_i h_i) = \max_G\{g,h\}_i \min_G\{g,h\}_i$.*

Proof. Observe that $\text{out}_\text{M}(s^{(i-1)}, g_i h_i)$ does not depend on bits $i+1, \ldots, B$. As $g_{1\ldots i}, h_{1\ldots i}$ are valid i-bit strings, we may thus w.l.o.g. assume that $B = i$. For symmetry reasons, it suffices to show the claim for the first output bit $\text{out}_\text{M}(s_\text{M}^{(B-1)}, g_B h_B)_1$ only; the other cases are analogous.

Using Lemma 8.4, Definition 8.1, and the definition of out, it is straightforward to verify that the claim holds for stable $g, h \in \{0,1\}^B$. Our task is to prove this equality also for the case where g or h contain a metastable bit. By Theorem 8.7, we have that $s^{(B-1)} = (\Diamond)_\text{M}(g_{1\ldots B-1}, h_{1\ldots B-1})$.

Let j be the minimum index such that $g_j = \text{M}$ or $h_j = \text{M}$. Again, for symmetry reasons, we may assume w.l.o.g. that $g_j = \text{M}$; the case $h_j = \text{M}$ is symmetric. If $g_{1\ldots j-1} \neq h_{1\ldots j-1}$, applying Lemma 8.4 shows that either (i) $s^{(i-1)} = 01$ ($g <_G h$) or (ii) $s^{(i-1)} = 10$ ($g >_G h$). Assume (i); (ii) is treated analogously. As the state 01 is absorbing, it follows that $s^{(B-1)} = 01$, regardless of the further bits of g and h. As $\text{out}(01, g_B h_B)_1 = h_B$ for all $g_B h_B \in \{0,1\}^2$, we conclude that $\text{out}_\text{M}(s^{(B-1)}, g_B h_B)_1 = h_B$, as desired.

Hence, suppose that $g_{1\ldots j-1} = h_{1\ldots j-1}$ for the remainder of the proof. We consider the case that $\text{par}(g_{1\ldots j-1}) = 0$ first, i.e., $s^{(j-1)} = 00 = s^{(0)}$. By Definitions 8.1 and 8.2, $g_{j\ldots B}, h_{j\ldots B} \in V_{B-j+1}$, so we may w.l.o.g. assume $j = 1$ in the following. If $B = 1$,

$$\text{out}_\text{M}(s_\text{M}^{(B-1)}, g_B h_B)_1 = \text{out}_\text{M}(00, \text{M}h_B)_1 = \begin{cases} 1 & \text{if } h_1 = 1 \\ \text{M} & \text{otherwise,} \end{cases}$$

which equals $\max_G\{g,h\}_B$ (we simply have a 1-bit code). If $B>1$, Observation 8.9 yields that $g_{2\dots B}=10\dots0$. We distinguish several cases.

$h_1=\mathrm{M}$: Then also $h_{2\dots B}=10\dots0$. Therefore $g_B=h_B$, $\mathrm{out}(s,g_Bh_B)_1=g_B=h_B$ for any $s\in\{0,1\}^2$, and

$$\mathrm{out}_{\mathrm{M}}(s_{\mathrm{M}}^{(B-1)},g_Bh_B)_1=g_B=h_B=\max\nolimits_G\{g,h\}_B\,.$$

$h_1=1$ and $B=2$: Thus, $g<_G h$, i.e., we need to output $h_B=\max_G\{g,h\}_B$. Consider the two stabilizations of g, i.e., 01 and 11. If the first bit of g is resolved to 0, we would end up with $s^{(B-1)}=s^{(1)}=01$, regardless of further bits. If it is resolved to 1, then $s^{(1)}=11$. Thus,

$$\begin{aligned}\mathrm{out}_{\mathrm{M}}(s^{(B-1)},g_Bh_B)_1&=\mathrm{out}_{\mathrm{M}}(01,1h_B)_1*\mathrm{out}_{\mathrm{M}}(11,1h_B)_1\\&=\underset{h_B\preceq h'_B\in\{0,1\}}{\ast}\{h'_B,\min\{1,h'_B\}\}\\&=\underset{h_B\preceq h'_B\in\{0,1\}}{\ast}\{h'_B\}=h_B\,.\end{aligned}$$

$h_1=1$ and $B>2$: Again, $h_B=\max_G\{g,h\}_B$. Consider the two stabilizations of g, i.e., $010\dots0$ and $110\dots0$. If the first bit of g is stabilized to 0, we end up with $s^{(B-1)}=s^{(1)}=01$, as 01 is an absorbing state. If it is stabilized to 1, then $s^{(1)}=11$. As $g_{2\dots B}=10\dots0$, for any $h\preceq h'\in\{0,1\}^B$, the state machine will end up in either state 00 (if $h'_{2\dots B}=10\dots0$) or state 01. Overall, we get that (i) $s^{(B-1)}=01$, (ii) $s^{(B-1)}=00*01=0\mathrm{M}$ and $h_{2\dots B}=1\dots0$, or (iii) $s^{(B-1)}=0\mathrm{M}$ and $h_{2\dots B}=1\dots0\mathrm{M}$ (cf. Table 8.1). If (i) applies, $\mathrm{out}(s_{\mathrm{M}}^{(B-1)},g_Bh_B)_1=h_B$. If (ii) applies, $\mathrm{out}_{\mathrm{M}}(s^{(B-1)},g_Bh_B)_1=g_B=h_B$. If (iii) applies, then

$$\begin{aligned}\mathrm{out}_{\mathrm{M}}(s^{(B-1)},g_Bh_B)_1&=\mathrm{out}_{\mathrm{M}}(00,0\mathrm{M})_1*\mathrm{out}_{\mathrm{M}}(01,0\mathrm{M})_1\\&=0*1*0*1=\mathrm{M}=h_B\,.\end{aligned}$$

$h_1=0$: This case is symmetric to the previous two: depending on how g is resolved, we end up with $s^{(1)}=10$ or $s^{(1)}=00$, and need to output g_B. Reasoning analogously, we see that indeed $\mathrm{out}_{\mathrm{M}}(s^{(B-1)},g_Bh_B)_1=g_B$.

It remains to consider $\mathrm{par}(g_{1,\dots,j-1})=1$. Then $s^{(j-1)}=11$. Noting that this reverses the roles of max and min, we reason analogously to the case of $\mathrm{par}(g_{1,\dots,j-1})=0$. □

Remarks:

- We have decomposed the task of computing the output into computing $s^{(i)}$, $i\in[B]$, and applying $\mathrm{out}_{\mathrm{M}}$.
- We have decomposed computing $s^{(i)}$ into applying $\diamond_{\mathrm{M}}$ $i-1$ times.
- As $\mathrm{out}_{\mathrm{M}}$ and $\diamond_{\mathrm{M}}$ can be implemented by constant-sized circuits, we get a circuit of asymptotically optimal size $\mathcal{O}(B)$ computing $\max_G\{g,h\}$ and $\min_G\{g,h\}$.
- However, our main goal was to find a circuit of low *depth* performing this computation. Applying $\diamond_{\mathrm{M}}$ would yield a circuit of depth $\Omega(B)$!
- This is where we shamelessly exploit the associativity of $\diamond_{\mathrm{M}}$.

8.3 Parallel Prefix Computation

As $\diamond_{\mathrm{M}}$ is associative, $s^{(B-1)}$ is computed by *any* binary tree for which the leaves are the inputs $g_i h_i$, $i \in \{1, \ldots, B-1\}$, and whose inner nodes are $\diamond_{\mathrm{M}}$ (sub)circuits. Using a balanced tree then results in depth $\lceil \log(B-1) \rceil$. However, we need to compute *all* $s^{(i)}$, $i \in [B]$. We could simply use B trees, whose total number of inner nodes would be

$$\sum_{i=0}^{B-1} i = \frac{(B-1)B}{2} \in \Theta(B^2)\,,$$

still resulting in a circuit of the same depth. We can do much better!

Theorem 8.11. *Given a circuit C implementing an associative operator $\odot\colon D \times D \to D$ and inputs $g_i \in D$, $i \in [2^b]$ for some $b \in \mathbb{N}$, there is a circuit of size $\mathcal{O}(2^b|C|)$ and depth $\mathcal{O}(bd(C))$ outputting for each $i \in [2^b] \setminus \{0\}$ the value $\bigodot_{j=0}^{i} g_i$ (where $\bigodot_0^0 g_i = g_0$).*

Proof. Our circuit will have two stages. The first stage (see Figure 8.2) is a balanced binary tree whose leaves are the 2^b inputs and whose non-leaf nodes are copies of C. Each node receives as input the outputs of its two children. Enumerating the leaves in DFS order, leaf $i \in [2^b]$ "outputs" its assigned input value g_i. By induction on decreasing depth in the tree, we get that each node outputs $(\bigodot)_{j=i_{\min}}^{i_{\max}} g_i$, where $i_{\min}$ and $i_{\max}$ are the smallest and largest leaf in the subtree of the node, respectively.

The second stage (see Figure 8.3) of the circuit receives all the computed values as input and computes all $\bigodot_{j=0}^{i} g_i$, $i \in [2^b]$, using a recursive scheme. We can describe the recursion again as a binary tree, with a one-to-one correspondence of nodes to the ones from the first stage. However, now outputs flow from non-leaf nodes to their children as inputs. For notational convenience, introduce the special symbol $\epsilon \notin D$ with the semantics $\epsilon \odot s = s$ for all $s \in D \cup \epsilon$, i.e., ϵ means "do nothing" (clearly, the extended operator remains associative). Moreover, denote for each non-leaf node by its left child the one traversed first in the DFS tour and refer to the other as right child. Provide to each node two inputs: the output o of the "left" (see figure) child node in the first stage and the output p of its parent, where the root receives ϵ as second input. With this notation, each non-leaf node now outputs p to its left child and $p \odot o$ to its right child. Finally, the leaf i outputs $p \odot g_i$.

We claim that i outputs $\bigodot_{j=0}^{i} g_i$. We prove the claim by induction on the depth of the tree. It is trivial for a tree of depth 0, hence assume it is correct for depth $d \in \mathbb{N}_0$ and consider a tree of depth $d+1$. Applying the induction hypothesis to the left child of the root, we see that leaf $i \in [2^{b-1}]$ outputs $\epsilon \odot \bigodot_{j=0}^{i} g_i = \bigodot_{j=0}^{i} g_i$; note that we exploited that (the extended) $\odot$ is associative here. Applying the induction hypothesis to the right child, we see that leaf $i \in [2^b] \setminus [2^{b-1}]$ outputs $(\bigodot_{j=0}^{2^{b-1}-1} g_i) \odot (\bigodot_{j=2^{b-1}}^{i} g_i) = \bigodot_{j=0}^{i} g_i$.

Apart from wires, our construction has at each non-leaf node of each of the two trees one copy of C, plus a copy of C at each leaf in the second tree, for a total of $3 \cdot 2^b - 2 \in \mathcal{O}(2^b)$ copies of $|C|$. The depth of both trees is b. The claims on size and depth of the circuit follow. □

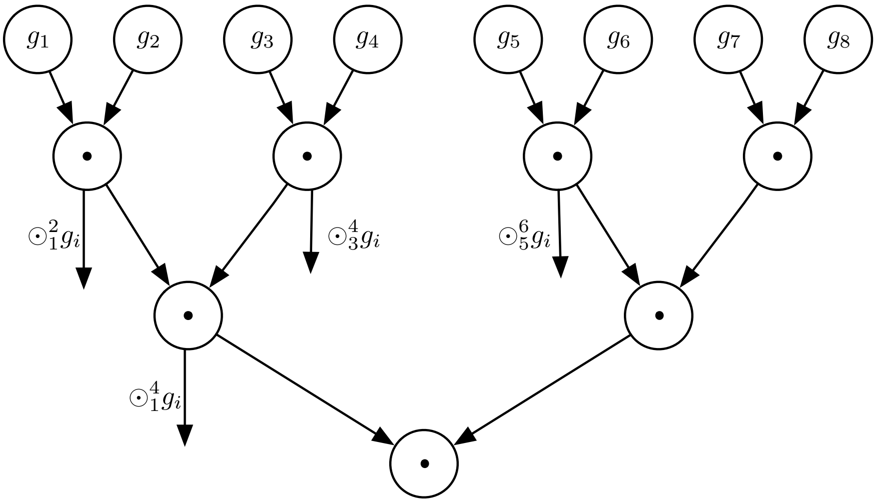


Figure 8.2: First stage of the construction in Theorem 8.11. Each tree node outputs the result of applying the operator to all leaves in its subtree. The output of the root and nodes reached from it by only "going to the right" are not needed; the nodes are there to show the tree structure.

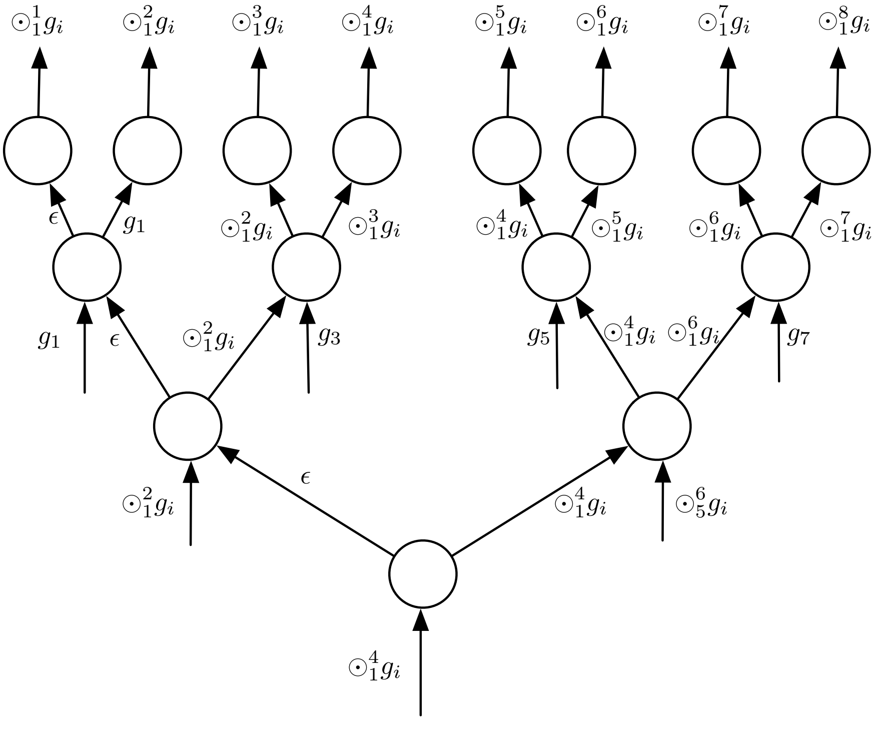


Figure 8.3: Second stage of the construction in Theorem 8.11. Using the outputs of the first stage, the nodes forward their input to left and the operator applied to the input from their parent and the one from the previous stage.

Corollary 8.12. *There is a comparator circuit of size $\mathcal{O}(B)$ and depth $\mathcal{O}(\log B)$ for valid strings (see Figure 8.4).*

Proof. By Theorem 6.6, there are circuits of constant size and depth that implement $\diamond_{\mathrm{M}}$ and $\mathrm{out}_{\mathrm{M}}$. We apply Theorem 8.11 to the circuit for $\diamond_{\mathrm{M}}$ and inputs $g_i h_i$, $i \in \{1, \dots, B-1\}$ (for $b = \lceil \log B \rceil$, simply ignoring the unneeded inputs and outputs to the circuit), yielding a circuit of size $\mathcal{O}(B)$ and depth $\mathcal{O}(\log B)$ computing outputs $s^{(i-1)}$, $i \in [B] \setminus \{0\}$. As $s^{(0)} = 00$ is a constant, we do not need a circuit to compute it. We then feed for $i \in \{1, \dots, B\}$ the inputs $s^{(i-1)}$ and $g_i h_i$ to a copy of the circuit implementing $\mathrm{out}_{\mathrm{M}}$, yielding the correct outputs. This adds $\mathcal{O}(B)$ to the size and increases the depth by a constant. □

Bibliographic Notes

There's almost nothing to add to the references given for the previous lecture. For the Parallel Prefix Computation (PPC) framework, see [LF80].

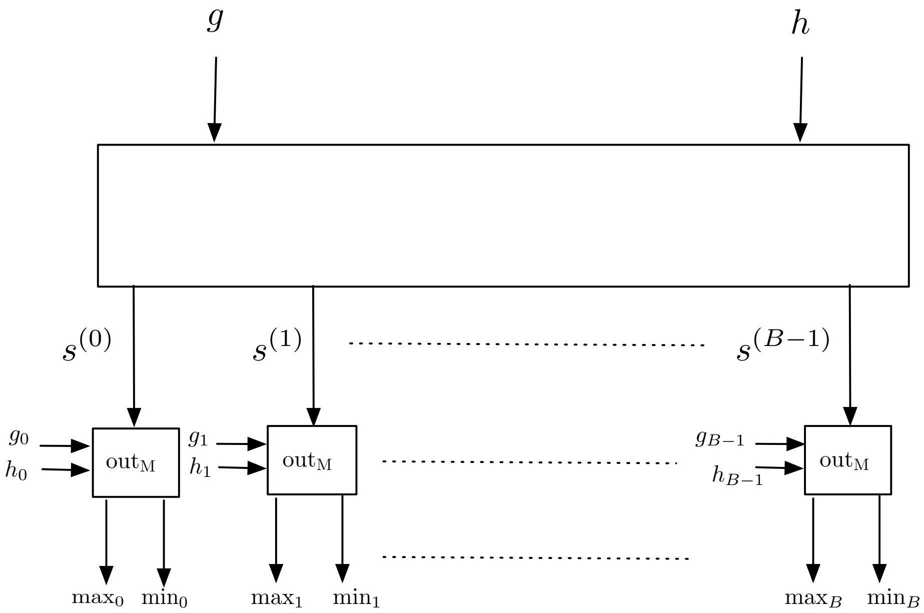


Figure 8.4: The Gray code comparator.

Bibliography

[LF80] Richard E Ladner and Michael J Fischer. Parallel Prefix Computation. *Journal of the ACM (JACM)*, 27(4):831–838, 1980.

Lecture 9

Self-Stabilization

So far we have considered permanently damaged (Byzantine) nodes. What if faults are transient? There are plenty of causes for such transient faults: radiation, power fluctuations, etc. One way of dealing with transient faults is to just consider the nodes undergoing faults becoming Byzantine, but then this may be to pessimistic: after the transient faults cease, they can recover a correct state and be good citizens again. Also, we may be able to recover from $n/3$ or more nodes undergoing transient faults. In fact, we want that the system recovers even if *all* nodes fail and $f < n/3$ of them *remain* faulty!

But what does "recover" from transient faults mean? We need to capture this in a way enabling us to prove (or disprove) this property for a given algorithm.

Definition 9.1 (Self-Stabilization). *Given a system, denote by S its state space, i.e., the possible values that transient memory of nodes, message buffers, and any other state-holding device in the system can hold. An* execution trace *is a path in S consistent with the obeying the restrictions the system model imposes on how the state may evolve over time. A* good trace *is one satisfying desired properties (depending on the task at hand). An algorithm is self-stabilizing, if it guarantees that any trace has a good suffix, i.e., any trace satisfies that there is a time such that its subtrace starting at this time is good. If this time until this subtrace starts is bounded, the stabilization time is a (possibly parametrized) worst-case upper bound on this time difference.*

Example 9.2 (Approximate Agreement (Flawed)). *Consider a synchronous system in which the nodes perform approximate agreement (Definition 5.2) using some algorithm $\mathcal{A}$. The state of the system is described by the subset of correct nodes $V_g \subseteq V$, their current values $r_v \in \mathbb{R}$, and whether they terminated, i.e., $S = \bigcup_{g=n-f}^{n}(\{0,1\} \times \mathbb{R})^g$ (plus any additional state the algorithm may maintain). A trace is an execution of the algorithm starting from any round r and state, which is completely defined by the messages faulty nodes send in rounds $i, i+1, \ldots$ A good trace is one in which all correct nodes eventually are terminated with values within ε of each other. No matter what $\mathcal{A}$ we choose, it will not be self-stabilizing: We choose as initial state one where all correct nodes are terminated, but their values are not within ε of each other. No correct node will change state anymore, so there is no good subtrace.*

Example 9.3 (Approximate Agreement (Fixed)). *Consider a synchronous system in which the nodes keep executing approximate agreement steps (i.e., per-*

form Algorithm 5.1) in each round. The state of the system is fully described by the subset of correct nodes $V_g \subseteq V$ and their current values $r_v \in \mathbb{R}$, i.e., $S = \bigcup_{g=n-f}^{n} \mathbb{R}^g$. A trace is an execution of the algorithm starting from any round i and state, which is completely defined by the messages faulty nodes send in rounds $i, i+1, \ldots$ We decide that a good trace satisfies that the diameter of all state vectors is smaller than ε and, in each round, values are within the range spanned by the correct nodes' values in the previous round. From Lemmas 5.4 and 5.5, we get that this holds for any round $j \geq i + \log(\|\vec{r}_i\|/\varepsilon)$.

Example 9.4 (GCS)**.** *Consider the task of gradient clock synchronization. The state space at time t is given by the nodes' hardware and logical clock values, their estimates of neighbors' clocks, the content of messages that are in transit and their sending times $t - d < t_s \leq t$, and any other state a node may hold according to the algorithm (none in case of our GCS algorithm — at least in the abstract model we considered!). A trace starting at time t is given by an* arbitrary *such state (even if it can't be reached in an execution faithful to the model!), from which we run the system in accordance with the model. A good trace starting at time t' is a trace satisfying a bound $\mathcal{L}$ on the local skew at all times $t'' \geq t'$. A self-stabilizing algorithm now guarantees that for any trace starting at time t, there is some time $t' \geq t$ so that the subtrace starting at time t' satisfies the local skew bound. The* stabilization time *of an algorithm is the maximum difference $t' - t$ over all traces (possibly parametrized by, e.g., the number of nodes n, etc.).*

Remarks:

- As the examples illustrate, the definition is quite flexible and can be applied to discrete and continuous systems, as well as those with and without permanent faults.
- "Time" is not clearly defined, as it depends on the system what this means. For example, in synchronous systems, time progresses discre(e)tely in rounds, while in GCS we have a continuous reference time.
- Even in the fixed approximate agreement example, the stabilization time is unbounded (i.e., ∞): the bound from Lemma 5.5 is tight in the worst case, and transient faults could bring the stored values arbitrarily far apart. We will show a stabilization time of $\mathcal{O}(\mathcal{G}/\mu)$ for the GCS algorithm. This is good in case there is a self-stabilizing mechanism ensuring a small global skew without interfering with the GCS algorithm (after stabilizing itself). If not, this is no better than the situation for approximate agreement: transient faults may bring the logical clocks arbitrarily far apart.
- It is important to carefully contemplate what "recovering correct operation" after transient faults actually means. This strongly affects whether a solution is possible and how efficient it can be. For instance, the approximate agreement example begs the question whether after transient faults, the (now arbitrarily corrupted) values nodes store hold any relevant information.
- The first example was bad because we asked for nodes to terminate. As we show below, self-stabilizing algorithms must never terminate, simply because a transient fault then could result in wrong output.

- The algorithm's code and the model assumptions are untouchable to transient faults. In the former case, that's obviously necessary: If transient faults can corrupt the algorithm itself, the algorithm designer has no chance to ensure recovery. It is thus advisable to hardwire the algorithm and/or store code in non-volatile memory. The model assumptions should be examined carefully, however. One will actually have to implement all this reliably, or the system might end up experiencing "transient" faults in perpetuity!

- For instance, this is relevant to the synchronous model. If the synchronous abstraction is implemented using an unreliable clocking method, a single "transient" fault may permanently disrupt the clocking scheme. Yes, the synchronous self-stabilizing algorithm will recover right after the clocking scheme — but the clocking scheme will never do so.

Lemma 9.5. *A self-stabilizing algorithm can never terminate, unless for each node there is a single output that is always correct.*

Proof. Suppose there are two possible conflicting outputs for a node v. More precisely, there is a terminal state (of the system as a whole) in which some possible differing terminal state of v is incorrect. We simply set the system to this state, but v to the incorrect one. As all nodes are in a terminal state, no further changes of node states is possible, implying that this combination of terminal states is preserved forever. Thus, the trace has no good suffix, showing that the algorithm is not self-stabilizing. □

Corollary 9.6. *Suppose* $\sigma = \mu/(\vartheta - 1) \in 1 + \Omega(1)$*. Then Algorithm 2.1 stabilizes in* $\mathcal{O}(\mathcal{G}/\mu)$ *time.*

Proof. W.l.o.g., consider traces that start at time 0. We claim that, for all $s \in \mathbb{N}$ and times $t \geq T_s := \sum_{s'=1}^{s} \mathcal{G}/(\mu\sigma^{s-1})$,

$$\Psi^s(t) \leq \frac{\mathcal{G}}{\sigma^s}\,.$$

The statement of the corollary then follows as in Theorem 2.9 for any time

$$t \geq \sup_{s\in\mathbb{N}}\{T_s\} = \sum_{s'=1}^{\infty} \frac{\mathcal{G}}{\mu\sigma^{s'-1}} = \frac{\sigma\mathcal{G}}{\mu(\sigma-1)} \in \mathcal{O}\left(\frac{\mathcal{G}}{\mu}\right)$$

due to the assumption on σ.

The claim is shown as for Theorem 2.16, with the modification that the time of violation now must be at least T_s, where s is the minimal level on which the bound is violated. This is only relevant in a single step of the proof, when invoking Lemma 2.15: here, the proof exploits that $\Psi^{s-1}(t_0) \leq t_1 - t_0 = \mathcal{G}/(\mu(\sigma^{s-1}))$. As $T_s - T_{s-1} = \mathcal{G}/(\mu(\sigma^{s-1}))$, $t_1 \geq T_s$ implies that $t_0 \geq T_{s-1}$, i.e., this condition is satisfied. □

Remarks:

- You might think “this was almost too easy.” The response to this has two parts. The first is that the algorithmic approach just happens to be that the algorithm continuously struggles on each level to distribute the skew in a way keeping Ψ^s small. The property of being self-stabilizing then emerges naturally. The second is that the model is hiding a lot of things. In order to make the algorithm self-stabilizing, one needs self-stabilizing solutions for maintaining a small global skew, computing estimates, and providing all the other convenient abstractions the model assumes.

- Fortunately, the algorithm really *is* doing a great job (which is rather coincidental, given that it was not designed with the goal of self-stabilization in mind). You’ll show in the exercises that the necessary adaptions are minimal.

9.1 Making Lynch-Welch Self-Stabilizing

Why the Lynch-Welch algorithm again? Well, it achieves asymptotically optimal skew, tolerates the maximum possible number of $\lceil n/3 \rceil - 1$ Byzantine faults, and it’s simple to implement. As we showed in the previous lectures, we can even handle metastability, which is a concern if we perform the iterations so quickly that it matters. Combining all of this with self-stabilization would result in an extremely robust algorithm!

Alas, we won’t get self-stabilization “for free” as with the GCS algorithm. The Lynch-Welch algorithm relies on some initial degree of synchronization to maintain the abstraction of rounds it uses. It is simulating synchronous execution, but self-stabilization requires that we can deal with a complete (initial) lack of synchrony! It turns out that this is an incredibly hard problem, and we will only take a first step today. This step is reducing the task to finding an (efficient) self-stabilizing solution to pulse synchronization with a much weaker bound on the skew.

Good traces are easily defined: There should be a time t from which on the algorithm behaves just like expected, i.e., as if it was initialized at this time and thus exhibits the skew and period bounds from Theorem 5.10.

9.2 First Attempt: Reset on Heartbeats

In the following, we assume that we already have a self-stabilizing pulse synchronization algorithm with skew σ_h in place. Thus, there is some time t when it stabilized from which on it generates pulses $h_{v,i}$, $v \in V_g$, $i \in \mathbb{N}$, satisfying that $\max_{i\in\mathbb{N}} \max_{v,w\in V_g}\{|h_{v,i} - h_{w,i}|\} \leq \sigma_h$. Moreover, we have lower and upper bounds on the time between pulses. We will refer to these pulses as *heartbeats,* or simply beats. They are supposed to be fairly slow in comparison to the pulses of the (modified) Lynch-Welch algorithm; from now on, when we talk of pulses, these will be those of the Lynch-Welch algorithm only.

There’s a single hurdle keeping the LW algorithm from being self-stabilizing: the need for a (known) bound on the initial deviation between the nodes’ local times. The heartbeats provide exactly that — they are at most σ_h apart from each other. So we could simply reset the LW algorithm on every heartbeat, setting $\mathcal{S} := \sigma_h$ for the initialization of the algorithm. That's going to work splendidly, as we won't even have to change the analysis — until the next beat comes along and messes things up. As the beats are not as well-synchronized as the LW pulses (otherwise we wouldn't go through this trouble), the reset will destroy the better synchronization guarantee again. Even worse, it may interrupt the LW algorithm generating a pulse!

Remarks:

- Actually, one needs to be slightly more careful when resetting, in that any messages sent by a node just before it is reset by its beat should not be confused with his "round 1"-message following the reset. This is easily addressed by offsetting the first round by ϑd local time compared to $h_{v,i}$, or by using the last message received during the time window in which receivers listen for messages from other nodes.

- It's important not to overlook such "details" when designing self-stabilizing algorithms. Another example is to make sure that variables that are stored in a way admitting infeasible values need to be regularly tested for having a valid value and reset to some default if not.

- It's also important to not lose sight of the big picture due to such details, though. A good way of designing self-stabilizing algorithms is to eliminate obstacles one by one, starting with establishing very basic properties and increasing the amount of "control" one has over the system state step by step. The more restricted the state beomes, the easier it typically becomes to reason about it and establish more complicated constraints.

- One could see what we're doing now as doing this process in reverse: We want to solve self-stabilizing pulse synchronization with asymptotically optimal skew, and reduce this task to solving self-stabilizing pulse synchronization with (fairly) large skew.

9.3 Second Attempt: Adding Feedback

The naive solution does not work, because heartbeats may arrive at inconvenient times. However, the "first" beat (in our analysis) establishes a timing relation between the LW instance and the instance of the self-stabilizing pulse synchronization algorithm generating the beats. If we add the additional requirement that the pulse synchronization algorithm accepts some external input that can shift the time when the next beat occurs (within certain bounds), we could align them with the pulses generated by the LW instance.

More specifically, after a (suitably chosen) fixed number of LW pulses, nodes will issue a NEXT signal to the part of them running the algorithm generating the heartbeats. Thus, the beat generation mechanism needs only be "responsive" to the NEXT signal within a specific time window in relation to the previous beat. Under some mild conditions on ϑ, this will turn out to be a fairly harmless constraint. We use this to trigger the next beat, aligned up to $\mathcal{O}(\sigma_h + \mathcal{S})$ time with when the nodes issue the NEXT signals (where $\mathcal{S}$ is the skew of the LW algorithm). This can be kept within a single round of the

LW algorithm (without affecting more than constants), as both $\sigma_h \in \mathcal{O}(d)$ and $\mathcal{S} \in \mathcal{O}(d)$, and the round duration of the LW algorithm $T \in \Omega(d)$ anyway.

Is this good enough? Not yet, as reset approach will cause large skew every time — unless, in addition, we require that well-synchronized NEXT signals result in an equally well-synchronized heartbeat. Instead of adding even more constraints on the self-stabilizing algorithm (not knowing whether they can be satisfied), we use a different solution.

9.4 Third Attempt: Reset on Unexpected Heartbeats Only

The final adjustment is to *not* perform a reset when a beat arrives on schedule, i.e., within a time window of size $\mathcal{O}(\sigma_h + \mathcal{S})$ around the point when it would occur in a world of perfect synchrony. The size of this window is chosen such that once the heartbeat generation has stabilized, after the first "proper" heartbeat it never happens again that a node is reset. Yet, the reset mechanism still guarantees that a heartbeat will enforce synchronization up to a skew of $\mathcal{O}(\sigma_h + \mathcal{S})$: either a node is not reset (defining a $\mathcal{O}(\sigma_h + \mathcal{S})$-sized window of possible local times) or it is (forcing the local time into the window).

It remains to formalize this approach and prove it correct. W.l.o.g., we assume in the following that the heartbeats stabilized by time 0, and start to reason from there. (Note, however, that this means that arbitrary messages may be in transit at time 0!). Let us first specify our expecations on the feedback mechanism.

Definition 9.7 (Feedback Mechanism)**.** *Nodes $v \in V_g$ generate* beats *at times $h_{v,i} \in \mathbb{R}$, $i \in \mathbb{N}$, such that for parameters $0 < B_1 < B_2 < B_3 \in \mathbb{R}$ the following properties hold for all $i \in \mathbb{N}$.*

1. *For all $v, w \in V_g$, we have that $|h_{v,i} - h_{w,i}| \leq \sigma_h$.*

2. *If no $v \in V_g$ triggers its NEXT signal during $[\min_{w \in V_g}\{h_{w,i}\} + B_1, t]$ for some $t < \min_{w \in C}\{h_{w,i}\} + B_3$, then $\min_{w \in V_g}\{h_{w,i+1}\} > t$.*

3. *If all $v \in V_g$ trigger their NEXT signals during $[\min_{w \in V_g}\{h_{w,i}\} + B_2, t]$ for some $t \leq \min_{w \in V_g}\{h_{w,i}\} + B_3$, then $\max_{w \in V_g}\{h_{w,i+1}\} \leq t + \sigma_h$.*

B_1, B_2, and B_3 cannot be chosen arbitrarily for our approach to work. We will determine sufficient constraints from the analysis.

In order to describe the algorithm, we assume that each node is running an instance of Algorithm 5, the beat generation algorithm, and some additional high-level control we give now. The high-level control may (re-)initialize the instance of Algorithm 5, which is described in the subroutine `reset`(τ) it may call. It has a few parameters:

M: The pulses of Algorithm 5 are counted modulo M. Every M pulses, a heartbeat is expected.

R^-: If a beat arrives at time t and the pulse number is 0 mod M, it should take at least R^- local time before the node generates the next pulse. Instead of trying to compute upfront when Algorithm 5 would generate a pulse, we simply "catch" the event and perform a reset if the pulse would be generated too early.

R^+: This is the matching upper bound, i.e., under the same conditions, it should take at most R^+ local time before the node generates the next pulse.

$\mathcal{S}(r)$: This denotes the skew bound guaranteed by Algorithm 5 for pulse $1 < r \in \mathbb{N}$, provided the algorithm is initialized with skew $\mathcal{S}(1)$, i.e., in the code of Algorithm 5, $\mathcal{S}$ is replaced by $\mathcal{S}(1)$. Algorithm 6 needs to make use of $\mathcal{S}(1)$ and $\mathcal{S}(M)$ only.

Algorithm 9.1: Interface algorithm, actions for node $v \in V_g$ in response to a local event at time t. Runs in parallel to local instances of the beat generation algorithm and Algorithm 5.

```
// algorithm maintains local variable $i \in [M]$
if $v$ generates a pulse at time $t$ then
    $i := i + 1 \bmod M$;
    if $i = 0$ then
        wait until local time $H_v(t) + \vartheta\mathcal{S}(M)$;
        trigger NEXT signal;
if $v$ generates a beat at time $t$ then
    if $i \neq 0$ then
        // beats should align with every $M^{th}$ pulse, hence reset
        reset($R^+$);
    else if Algorithm 5 would require $v$ to generate a pulse before local
      time $H_v(t) + R^-$ then
        // reset to avoid early pulse or message
        reset($R^+ - (H_v(t') - H_v(t))$), where $t'$ is the current time;
    else if next pulse is not generated by local time $H_v(t) + R^+$ then
        // reset to avoid late pulse and start listening for other nodes'
          pulses on time
        reset(0);
Function reset($\tau$)
    halt local instance of Algorithm 5;
    wait for $\tau$ local time;
    $i := 0$;
    $L_v(t') := \mathcal{S}(1)$, where $t'$ is current time;
    generate pulse and restart loop of Algorithm 5 (in round $r = 1$);
```

Figure 9.1 illustrates how the control algorithm ensures stabilization. In words, Algorithm 6 triggers the NEXT signal $\vartheta\mathcal{S}(M)$ local time after generating a beat (i.e., at the earliest time when certainly all nodes have generated the beat), and checks whether pulse 1 modulo M occurs between R^- and R^+ local time after the beat (which necessitates that the beat occurs after pulse 0 modulo M). If this is not the case, the algorithm generates a pulse and restarts the loop of Algorithm 5 exactly R^+ local time after the beat was generated. Moverover, it ensures that no other pulse is generated between the beat and then.

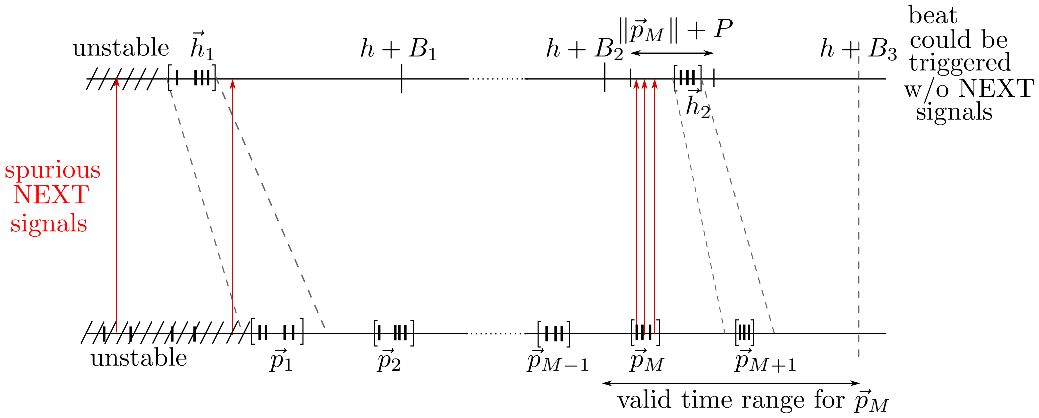


Figure 9.1: Interaction of the beat generation and Algorithm 5 in the stabilization process, controlled by Algorithm 6. Beat $\vec{h}_1$ forces pulse $\vec{p}_1$ to be roughly synchronized. The approximate agreement steps then result in tightly synchronized pulses. By the time the nodes trigger beat $\vec{h}_2$ by providing NEXT signals based on $\vec{p}_M$, synchronization is tight enough to guarantee that the beat results in no resets.

This properly "initializes" Algorithm 5 with skew $\mathcal{S}(1) := R^+ + \sigma_h - R^-/\vartheta$, which then ensures that the skew has been reduced to $\mathcal{S}(M)$ by the time the next beat is due. By choosing all parameters right, we ensure that the M^{th} pulse (after stabilization) falls in the time window provided by Definition 9.7 for making use of the NEXT signals, which then trigger the next beat such that no $v \in V_g$ performs another reset. From there, inductive reasoning shows that no $v \in V_g$ ever performs a reset again (so as long as there are no more transient faults), and the analyis of Algorithm 5 from Chapter 5 yields a bound on the skew achieved. Figure 9.2 illustrates how the nodes locally check whether they should perform a reset or not.

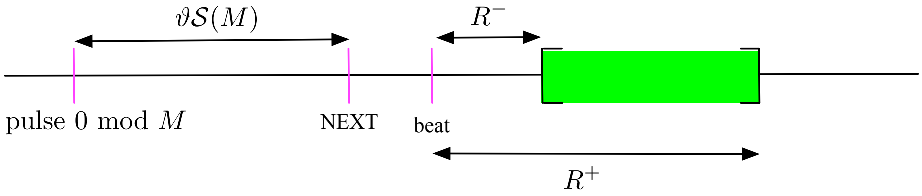


Figure 9.2: After M pulses a node v waits for $\mathcal{S}(M)$ local time and then generates the NEXT signal. After stabilization, the next heartbeat occurs shortly after. If the next pulse (which is going to be generated by the Lynch-Welch algorithm), with number $i = 1$, is not generated at least R^- and at most R^+ local time after the heartbeat (the green box), the node resets the Lynch-Welch algorithm, restarting its loop R^+ local time after the beat.

9.5 Analysis

In the following, we assume that in Algorithm 5, $\mathcal{S}$ is replaced by $\mathcal{S}(1)$ in the code, estimates are computed according to Lemma 5.9 (yielding $\delta = u + (\vartheta - 1)d + 2(\vartheta^2 - \vartheta)\mathcal{S}(1)$), and $T := \vartheta((\vartheta^2 + \vartheta + 1)\mathcal{S}(1) + \vartheta d)$ (in accordance with Lemma 5.8); as we require that $\mathcal{S}(1) \geq \mathcal{S}(r)$ for all $r \in \mathbb{N}$ (which is implied by (9.8)), this means that T is large enough for all rounds. For the outlined approach to work in addition the following constraints need to be satisfied.

$$\mathcal{S}(1) \geq 2\left(\delta + \left(1 - \frac{1}{\vartheta}\right) T\right) \tag{9.1}$$

$$\frac{R^-}{\vartheta} \geq \sigma_h + \vartheta \mathcal{S}(1) + d \tag{9.2}$$

$$\frac{B_2}{\vartheta} > \sigma_h + R^+ + T + 2\mathcal{S}(1) \tag{9.3}$$

$$B_1 > \sigma_h + R^+ \tag{9.4}$$

$$B_3 > R^+ + (M-1)(T + \mathcal{S}(1)) + (\vartheta + 1)\mathcal{S}(M) + \sigma_h \tag{9.5}$$

$$B_2 \leq \frac{R^-}{\vartheta} + (M-1)\left(\frac{T}{\vartheta} - \mathcal{S}(1)\right) + \mathcal{S}(M) \tag{9.6}$$

$$\frac{R^+}{\vartheta} \geq (\vartheta + 1)\mathcal{S}(M) + \sigma_h \tag{9.7}$$

$$2(\mathcal{S}(1) - \mathcal{S}(M)) \geq \sigma_h \tag{9.8}$$

We will worry later about satisfying all of these constraints. For now, we assume that they hold; what follows is conditional on this assumption.

We first establish that the first beat guarantees to "initialize" the synchronization algorithm such that it will run correctly from this point on (neglecting for the moment the possible intervention by further beats). We use this do define the "first" pulse times $p_{v,1}$, $v \in V_g$, as well; we enumerate consecutive pulses accordingly.

Lemma 9.8. *Let $h := \min_{v \in V_g}\{h_{v,1}\}$ and $\mathcal{S}(1) := R^+ + \sigma_h - R^-/\vartheta$. We have that*

1. *Each $v \in V_g$ generates a pulse at a unique time $p_{v,1} \in [h + R^-/\vartheta, h + \sigma_h + R^+]$.*
2. *$\|\vec{p}(1)\| \leq \mathcal{S}(1)$.*
3. *At time $p_{v,1}$, $v \in V_g$ sets $i := 1$.*
4. *At the time $\min_{v \in V_g}\{p_{v,1}\}$, no message (of Algorithm 5) sent by node $v \in V_g$ before time $p_{v,1}$ is in transit any more.*

Proof. Assume for the moment that $\min_{v \in V_g}\{h_{v,2}\}$ is sufficiently large, i.e., no second beat will occur at any correct node for the times relevant to the proof of the lemma; we will verify this at the end of the proof.

From the pseudocode given in Algorithm 6, it is straightforward to verify that $v \in V_g$ generates a pulse at a local time from $[H_v(h_{v,1}) + R^-, H_v(h_{v,1}) + R^+]$, and does not generate a pulse at a local time from $[H_v(h_{v,1}), H_v(h_{v,1}) + R^-)$. By Algorithm 5 and the choice of $\mathcal{S}(1)$, no $v \in V_g$ will send a message or generate another pulse during $[p_{v,1}, p_{v,1} + \mathcal{S}(1)]$, where $p_{v,1} \geq h_{v,1} + R^-/\vartheta + \mathcal{S}(1) \geq h + \sigma_h + R^+$. Since $h_{v,1} \in [h, h + \sigma_h]$ for all $v \in V_g$ by Definition 9.7, hence the times $p_{v,1} \in [h + R^-/\vartheta, h + \sigma_h + R^+]$, $v \in V_g$, are indeed unique. The second claim is now immediate from the choice of $\mathcal{S}(1)$.

Concerning the third claim, observe that if at time $h_{v,1}$ it held that the i-variable of $v \in V_g$ was not 0, it was set to 0. Thus, when v generates its next pulse at time $p_{v,1}$, it is increased to 1. Concerning the final claim, we have established that $v \in V$ generates no pulse during $[h + \sigma_h, h + R^-/\vartheta)$; thus, it sends no message during $[h + \sigma_h + \vartheta\mathcal{S}(1), h + R^-/\vartheta)$ (cf. Algorithm 5), and Inequality (9.2) ensures that no message of $v \in V_g$ sent before time $h_{v,1}$ is in transit any more at time $p_{w,1}$ for any $w \in V_g$.

It remains to show that indeed $\min_{v \in V_g}\{h_{v,2}\}$ is sufficiently large to not interfere with the above reasoning. Clearly, this is the case if round 1 ends at all nodes before this time. Let H be infimal with the property that any $v \in V_g$ executes `reset` at a time larger that $p_{v,1}$. Clearly, $H \geq \min_{v \in V_g}\{h_{v,2}\}$. By Definition 9.7 and Inequality (9.3), we can conclude that $H \geq h + B_2 \geq h + \sigma_h + R^+ + T + 2\mathcal{S}(1)$. All parts of the statements of this lemma that refer to times smaller than H hold. As $H > h + \sigma_h + R^+$, this implies that Algorithm 5 behaves exactly as if it was initialized with skew $\mathcal{S}(1)$ at time $h + R^-/\vartheta$. We can thus apply all results from Chapter 5 (for times $t < H$) accordingly. In particular, we get the same results as in Theorem 5.10 (as Inequality (9.1) and our choice of T and δ make sure that we can apply all lemmas), yielding that

$$\max_{v \in V_g}\{p_{v,2}\} \leq \min_{v \in V_g}\{p_{v,1}\} + P_{\max} \leq h + \sigma_h + R^+ + T + 2\mathcal{S}(1) < H\,. \qquad \square$$

Lemma 9.8 serves as induction anchor for the argument showing that all rounds of the algorithm are executed correctly. Let H be defined as in the previous proof. From the results in Chapter 5, we can bound $\mathcal{S}(r)$ for rounds $r \in \mathbb{N}$ that are complete before time H.

Corollary 9.9. *Suppose for $r \in \mathbb{N}$ that $\max_{v \in V_g}\{p_{v,r}\} < H$. Then*

$$\begin{aligned}
\|\vec{p}_r\| &\leq \mathcal{S}(r)\\
&:= \frac{\mathcal{S}(1)}{2^{r-1}} + \left(2 - \frac{1}{2^{r-2}}\right)\left(\delta + \left(1 - \frac{1}{\vartheta}\right)T\right)\\
&< \frac{\mathcal{S}(1)}{2^{r-1}} + 2(u + (\vartheta^2 - 1)d + (\vartheta - 1)(\vartheta^2 + 3\vartheta + 1)\mathcal{S}(1))\,,
\end{aligned}$$

which for sufficiently large $r \in \mathbb{N}$ is in $\mathcal{O}(u + (\vartheta - 1)(d + \mathcal{S}(1)))$. Moreover, the generated pulses satisfy $P_{\min} \geq T/\vartheta - 2\mathcal{S}(1)$ and $P_{\max} \leq T + 2\mathcal{S}(1)$.

Proof. We inductively apply Lemmas 5.8, and 5.5, yielding

$$\begin{aligned}
\|\vec{p}_r\| &\leq \frac{\mathcal{S}(1)}{2^{r-1}} + \sum_{r'=2}^{r} \frac{1}{2^{r-r'}}\left(\delta + \left(1 - \frac{1}{\vartheta}\right)T\right)\\
&= \frac{\mathcal{S}(1)}{2^{r-1}} + \left(2 - \frac{1}{2^{r-2}}\right)\left(\delta + \left(1 - \frac{1}{\vartheta}\right)T\right)\,.
\end{aligned}$$

Plugging in δ and our choice of T and bounding $2 - 2^{-(r-2)} < 2$ yields the stated upper bound on this term. By Inequality (9.8), we have that $\mathcal{S}(1) \geq \mathcal{S}(r)$ for all $r \in \mathbb{N}$. Thus, the inductive use of the lemmas (cf. Statement (iii) of Lemma 5.8) also shows the bounds on the period. $\square$

In other words, all we need to show is that $H = \infty$, i.e., no further resets occur after the first beat. In fact, it suffices to show this for the second beat, as this constitutes the necessary induction step. To this end, we first show that the NEXT signals occur within the "window of opportunity" provided by Definition 9.7.

Lemma 9.10. *For all $v \in V_g$, it holds that $h_{v,2} \in (p_{v,M} + \mathcal{S}(M), p_{v,M} + (\vartheta + 1)\mathcal{S}(M) + \sigma_h]$. In particular, no node calls the* `reset` *subroutine due to its second beat.*

Proof. Checking Algorithm 6 (and noting that by Lemma 9.8 we have that i is set to 1 at time $p_{v,1}$), we see that after time $p_{v,1}$, $v \in V_g$ will not locally trigger a NEXT signal before either time $p_{v,M} + \mathcal{S}(M)$ or H. Denote $p := \min_{v\in V_g}\{p_{v,M}\}$. As Lemma 9.8 and Inequality (9.4) show that $\max_{v\in V_g}\{p_{v,1}\} \leq h + \sigma_h + R^+ \leq h + B_1$, no NEXT signal is triggered during $[h+B_1, \min\{p+\mathcal{S}(M), H\}]$. However, by Definition 9.7, in absence of any NEXT signal, $h' := \min_{v\in V_g}\{h_{v,2}\}$ satisfies $h' \geq h + B_3$, implying that no NEXT signal is triggered during $[h+B_1, \min\{p+\mathcal{S}(M), h+B_3\}]$. By Definition 9.7, this entails that $H \geq h' \geq \min\{p+\mathcal{S}(M), h+B_3\}$, where equality can hold only if $h' = h + B_3$.

Next, we show that $h' < h + B_3$. Assuming the contrary, we have that $H \geq h' \geq h + B_3$, and get from Lemma 9.8 and Corollary 9.9 that

$$\begin{aligned}
&p + (\vartheta + 1)\mathcal{S}(M) + \sigma_h \\
&\leq \min\left\{\max_{v\in V_g}\{p_{v,1}\} + (M-1)(T + \mathcal{S}(1)) + (\vartheta+1)\mathcal{S}(M) + \sigma_h, H\right\} \\
&\leq \min\left\{h + \sigma_h + R^+ + (M-1)(T + \mathcal{S}(1)) + (\vartheta+1)\mathcal{S}(M) + \sigma_h, h + B_3\right\} \\
&< h + B_3\,,
\end{aligned}$$

where the last step uses Inequality (9.5). Thus, as H is larger than this time, each $v \in V_g$ triggers its NEXT signal before time $h + B_3 - \sigma_h$, because the corollary also shows that $\max_{v\in V_g}\{p_{v,M}\} \leq p + \mathcal{S}(M)$, and nodes wait for $\vartheta\mathcal{S}(M)$ local time before triggering the signal. On the other hand, Lemma 9.8, Corollary 9.9, and Inequality (9.6) show that

$$\begin{aligned}
p + \mathcal{S}(M) &\geq \min_{v\in V_g}\{p_{v,1}\} + (M-1)\left(\frac{T}{\vartheta} - \mathcal{S}(1)\right) + \mathcal{S}(M) \\
&\geq h + \frac{R^-}{\vartheta} + (M-1)\left(\frac{T}{\vartheta} - \mathcal{S}(1)\right) + \mathcal{S}(M) \\
&\geq h + B_2\,,
\end{aligned}$$

i.e., all of these NEXT signals are triggered no earlier than time $h + B_2$. By Definition 9.7, this entails that $h' \leq p + (\vartheta+1)\mathcal{S}(M) + \sigma_h < h + B_3$, contradicting the assumption that $h' \geq h + B_3$.

Knowing that $h' < h + B_3$, we can conclude that $\max_{v\in V_g}\{p_{v,M}\} \leq p + \mathcal{S}(M) < h' \leq H$. As we can derive the same bounds as above, we also get that $\max_{v\in V_g}\{h_{v,2}\} \leq p + (\vartheta+1)\mathcal{S}(M) + \sigma_h = \min_{v\in V_g}\{p_{v,M}\} + (\vartheta+1)\mathcal{S}(M) + \sigma_h$, provided that no node performs a reset before triggering its NEXT signal, i.e., $H > p + (\vartheta + 1)\mathcal{S}(M) + \sigma_h$. Recalling that we already established that $H \geq h' > \max_{v\in V_g}\{p_{v,M}\}$, the local i variables have been set to 0 mod M again, and will not change before the next pulse. Checking Algorithm 6, we see that such a reset thus would either occur R^+ local time after the (local) beat or due to the next pulse occuring before local time $h_{v,2} + R^-$. As $R^+/\vartheta \geq (\vartheta+1)\mathcal{S}(M) + \sigma_h$ by Inequality (9.7), the former cannot happen.

Observe that if the latter does not take place either, it would indeed follow that no node performs a reset on its second beat. Therefore, we conclude that $H \geq \min_{v \in V_g}\{p_{v,M+1}, p + (\vartheta+1)\mathcal{S}(M) + \sigma_h\}$ (where we slightly abuse notation in that if v *would* generate pulse $M+1$, but Algorithm 6 prevents this and performs a reset instead, we still denote this time by $p_{v,M+1}$). Finally, assume for contradiction that $H < p + (\vartheta+1)\mathcal{S}(M) + \sigma_h$. Thus, there is some $v \in V_g$ so that $H = p_{v,M+1} < p + (\vartheta+1)\mathcal{S}(M) + \sigma_h$. However, as v is the first node performing a reset, the period bound applies, i.e.,

$$\begin{aligned} p_{v,M+1} &\geq p + \frac{T}{\vartheta} - \mathcal{S}(1) \\ &= p + (\vartheta^2 + \vartheta)\mathcal{S}(1) + \vartheta d \\ &> p + (\vartheta+1)\mathcal{S}(M) + 2(\mathcal{S}(1) - \mathcal{S}(M)) \\ &\geq p + (\vartheta+1)\mathcal{S}(M) + \sigma_h\,, \end{aligned}$$

where the last step uses Inequality (9.8). Thus all possible cases lead to the desired bounds on $h_{v,2}$ for all $v \in V_g$. □

We summarize today's findings in the following theorem.

Theorem 9.11. *Assume that* $3+4\vartheta-4\vartheta^2-2\vartheta^3 > 0$ *and Inequalities* (9.1)-(9.8) *hold. Set* $T := \vartheta((\vartheta^2+\vartheta+1)\mathcal{S}(1) + \vartheta d)$*, where* $\mathcal{S}(1) := R^+ + \sigma_h - R^-/\vartheta$*. If the beats behave as required by Definition 9.7, Algorithm 6 running in conjunction with Algorithm 5 (where estimates are computed according to Lemma 5.9) is a self-stabilizing solution to the pulse synchronization problem. Its skew is in* $\mathcal{O}(u + (\vartheta-1)(d+\mathcal{S}(1)))$ *and the generated pulses satisfy* $P_{\min} \geq T/\vartheta - 2\mathcal{S}(1)$ *and* $P_{\max} \leq T + 2\mathcal{S}(1)$*. The stabilization time (not accounting for the beats) is* $\mathcal{O}(MT)$*.*

Proof. We apply Lemma 9.10 to each beat but the first, showing that $H = \infty$. Corollary 9.9 then yields the claims. □

Bibliographic Notes

The concept of self-stabilization was introduced by Dijkstra [Dij74]. The definition here is more general, but in turn also somewhat informal — notions like "time" need to be assigned meaning according to the specific system model. There are some generic constructions for self-stabilizing algorithms. For instance, Awerbuch et al. showed that any synchronous message-passing algorithm can be modified into a self-stabilizing asynchronous message-passing algorithm that stabilizes in the same time as needed to compute the solution from scratch [APSV91]. The approach for making the Lynch-Welch algorithm self-stabilizing discussed in this lecture is taken from [KL18].

Bibliography

[APSV91] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-Stabilization By Local Checking and Correction. In *In Proceedings* *of IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):943–644, November 1974.

[KL18] Pankaj Khanchandani and Christoph Lenzen. Self-Stabilizing Byzantine Clock Synchronization with Optimal Precision. *Theory of Computing Systems*, 2018.

Lecture 10

Consensus

One of the key components in the self-stabilizing pulse synchronization algorithm we will see two lectures from now is *consensus.* Consensus is a fundamental and extremely well-studied fault-tolerance primitive. There are a large number of variants of the problem, varying in terms of the model and the requirements on the solution. The common theme is the following question: In a system with faults, how can the non-faulty nodes agree on a decision that is consistent with given inputs?

Today, we study a very basic formulation of the consensus problem. We assume a synchronous system (like in Chapter 5 for approximate agreement) with n nodes and $f < n/3$ Byzantine faults, where nodes have unique identifiers $1, \ldots, n$ known to all nodes. Each node is given a binary input $b_i \in \{0, 1\}$. To solve (binary) consensus, an algorithm must compute output values $o_i \in \{0, 1\}$ at all correct nodes $i \in V_g$ meeting the following conditions:

Agreement: There is $o \in \{0, 1\}$ so that $o_i = o$ for all $i \in V_g$. We refer to o as the output of the consensus algorithm.

Validity: If there is $b \in \{0, 1\}$ so that for all $i \in V_g$ it holds that $b_i = b$, then $o = b$.

Termination: There is $r \in \mathbb{N}$ satisfying that each $i \in V_g$ terminates and outputs o_i by the end of round r.

In general, the round r when all correct nodes have terminated may depend on the execution. However, we are interested in algorithms which guarantee termination within $R(f) \in \mathbb{N}$ rounds, regardless of the inputs and the behavior of faulty nodes. We refer to $R(f)$ as the *running time* or *round complexity* of the algorithm.

Remarks:

- For $f \geq n/3$, no algorithm solves consensus deterministically. The reasons are very similar to what we saw in Chapter 4.
- Even randomized algorithms fail with a large probability. One could say that the $n/3$ barrier is "hard." However, under cryptographic assumptions one can "force" faulty nodes to communicate consistently (either not sending a message or sending the same to everyone), so long as $f < n/2$. Then,

in the synchronous model, the task is trivial: All nodes broadcast their input, and choose the output as a function of the received values ensuring validity.

- Beside the round complexity, we care about the amount of communication. Relevant criteria here are the maximum message size (i.e., the number of bits in the largest message an algorithm uses) and the number of bits nodes send in total.

10.1 The Phase King Algorithm

Algorithm 10.1: Phase King Algorithm at node $i \in V_g$. Each broadcast takes one round. Note that faulty nodes are not required to broadcast, i.e., they can send conflicting messages to different nodes.

```
$op_i := b_i$
for $j = 1 \dots f+1$ do
    // first broadcast
    strong := 0
    broadcast $op_i$ (also to self)
    if received at least $n - f$ times $b \in \{0,1\}$ then
        $op_i := b$
        strong := 1
    // second broadcast
    if strong = 1 then
        broadcast $op_i$
    if received fewer than $n - f$ times $op_i$ then
        strong := 0
    // king's broadcast part I
    if $i = j$ and received at least $f + 1$ times $b \in \{0,1\}$ then
        broadcast $b$
    // king's broadcast part II
    if $i = j$ and received at most $f$ times $b \in \{0,1\}$ then
        broadcast $op_i$
    // if not sure, obey the king
    if strong = 0 and received $b \in \{0,1\}$ from node $j$ then
        $op_i := b$
return $op_i$
```

In the above algorithm, we refer to one iteration of the loop as a *phase.*

Lemma 10.1. *If, for some $b \in \{0,1\}$ and all $i \in V_g$, $op_i = b$ at the beginning of the phase, then the same holds at the end of the phase.*

Proof. As $|V_g| \geq n - f$, each $i \in V_g$ will not change op_i and set strong to 1 after the first broadcast. Thus, in the second broadcast, $|V_g| \geq n - f$ nodes will broadcast b, and all correct nodes will maintain strong = 1. Thus, op_i is not changed by the king's broadcast. □

Corollary 10.2. *Algorithm 10.1 satisfies validity.*

Proof. Suppose $b_i = b$ for some $b \in \{0, 1\}$ and all $i \in V_g$. Then each $i \in V_g$ initializes $op_i := b$, which by inductive use of Lemma 10.1 never changes. Thus each $i \in V_g$ outputs b. □

Lemma 10.3. *Suppose node $j \in V_g$. Then there is some $b \in \{0, 1\}$ so that $op_i = b$ for all $i \in V_g$ at the end of phase j.*

Proof. Fix the phase to be j. We claim that there is $b \in \{0, 1\}$ satisfying that each $i \in V_g$ with strong $= 1$ after the first broadcast satisfies $op_i = b$. Otherwise, as correct nodes broadcast, it would hold that

$$2n - 2f = 2(n - f) \leq |V_g| + 2(n - |V_g|) = 2n - |V_g| ,$$

i.e., $|V_g| \leq 2f < n - f$, as $f < n/3$—which implied that there are $n - |V_g| > f$ faulty nodes.

Thus, only faulty nodes may send a value different from b in the second broadcast. We distinguish two cases. The first is that there is no $i \in V_g$ with strong $= 1$ after the second broadcast. In this case, each $i \in V_g$ sets $op_i := b' \in \{0, 1\}$, where b' is the value broadcasted by the king, i.e., node j.

The other case is that some node received $n - f$ times b in the second broadcast, the king (i.e., node j) received at least $n - 2f \geq f + 1$ times b. On the other hand, there are at most f faulty nodes, so the king did not receive more than f times $1 - b$. It follows that the king broadcasts b in the king's broadcast. As $f < n - f$, any $i \in V_g$ with $op_i = 1 - b$ satisfies that strong $= 0$ when receiving this message and sets $op_i := b$. □

Corollary 10.4. *Algorithm 10.1 satisfies agreement.*

Proof. As there are at most f faults, $[f + 2] \cap V_g \neq \emptyset$. Let $j \in [f + 2] \cap V_g$. By Lemma 10.3, at the end of phase j, we have that there is some $b \in \{0, 1\}$ so that $op_i = b$ for all $i \in V_g$. By inductive use of Lemma 10.1, these variables do not change any more. Hence all $i \in V_g$ output b. □

Theorem 10.5. *Algorithm 10.1 solves binary consensus in the synchronous model. It runs for $R(f) = 3(f+1) \in O(f)$ rounds and broadcasts 1-bit messages.*

Proof. Agreement and validity hold by Corollary 10.4 and Corollary 10.2, respectively. The running time bound follows from the facts that each phase takes three rounds, one for each broadcast, and that there are $f + 1$ phases. The message size bound is immediate from the pseudocode. □

Remarks:

- Depending on the precise model of communication, message size may also be 2 bits; the bound above exploits the option of sending no message, which may not always be possible.
- The message size is trivially optimal—but that does not mean the *overall* number of communicated bits is.
- Deterministic algorithms need to send $\Omega(nf)$ bits, which follows from the simple observation that each correct node needs to receive more than f bits to even know that anyone else would pick a certain output value.

- The running time is asymptotically optimal: no algorithm can be faster than $f+1$ rounds in the worst case. We will show this later in this lecture.
- Both the bit complexity and running time lower bound can be beaten by randomized algorithms; this is beyond the scope of this lecture, though.

10.2 Recursive Phase King

The Phase King protocol is doing well in terms of running time and resilience, i.e., f. However, it uses much more communication than the (trivial) lower bound of $\Omega(nf)$ requires. We can overcome this by avoiding to have all nodes communicate to each other for f times. The key to achieving this is to make sure that the "king" is more likely to be reliable. We do this by calling the protocol on roughly half of the participating nodes. There are not enough faulty nodes to make both instances fail, and in the recursive calls, fewer nodes need to send and receive messages. In the following recursive variant of the protocol, "broadcast" means to sent a message to all nodes participating in the instance.

Lemma 10.6. *Algorithm 10.2 satisfies agreement and validity.*

Proof. We reason similarly to our analysis of Algorithm 10.1, with the difference that the role of the "king" is now filled by V'. We show the claim by induction on $|V|$; it is trivial for $|V| = 1$. For $|V| > 1$, observe that the statement of Lemma 10.1 can be shown analogously for Algorithm 10.2.

Next, note that f is the number of faults we expect Algorithm 10.2 to withstand. If $|V| > 1$, define that a recursive call is successful if $|V'| > 3|V' \setminus V_g|$, i.e., there are few enough faults such that validity and agreement hold for the recursive call by the induction hypothesis. We show that at least one of the two recursive calls is successful. To this end, recall that for $j = 1$ V' is chosen such that the recursive call is successful, if $|V' \setminus V_g| \leq \lceil (f-1)/2 \rceil$. Concerning $f = 2$, note that

$$3\left\lceil \frac{f-1}{2} \right\rceil + 1 + 3\left\lfloor \frac{f-1}{2} \right\rfloor + 1 = 3(f-1) + 2 < 3f < n\,.$$

Hence, the second recursive call is successful, if $|V' \setminus V_g| \leq \lfloor (f-1)/2 \rfloor$. Overall, as there are at most

$$f < f+1 = \left\lceil \frac{f-1}{2} \right\rceil + 1 + \left\lfloor \frac{f-1}{2} \right\rfloor + 1$$

faults, at least one of the recursive calls is successful.

Now recall Lemma 10.3. Instead of requiring that $j \in V_g$, we instead demand that the j^{th} recursive call succeeds. We proceed as in the proof of Lemma 10.3 until the case distinction. If there is no correct node with strong = 1, then by the agreement property, all correct nodes in V' broadcast the same value $b \in \{0,1\}$, and because $|V'| > 3|V' \setminus V_g|$, this is for each node the majority value received from nodes in V'. Thus, each node sets $op_i = b$, as desired. On the other hand, if there is a correct node i with strong = 1, then the same reasoning as in Lemma 10.3 shows that each node in $V' \cap V_g$ uses input op_i for the recursive call. By validity, this means that this is the output of the recursive call, which is broadcasted to all nodes by the majority of nodes in

Algorithm 10.2: Recursive Phase King Algorithm at node $i \in V_g$. For simplicity, recursive calls on k nodes assume the identifiers to be $\{1, \dots, k\}$.

```
if $|V| = 1$ then
    return $b_i$
$op_i := b_i$
for $j = 1, 2$ do
    strong := 0
    broadcast $op_i$ (also to self)
    if received at least $n - f$ times $b \in \{0, 1\}$ then
        $op_i := b$
        strong := 1
    if strong = 1 then
        broadcast $op_i$
    if received fewer than $n - f$ times $op_i$ then
        strong := 0
    // recursive call
    $f := \lceil n/3 \rceil - 1$
    if $j = 1$ then
        $V' := \{1, \dots, 3\lceil (f-1)/2 \rceil + 1\}$
    else
        $V' := \{3\lceil (f-1)/2 \rceil + 2, \dots, |V|\}$
    if $i \in V'$ then
        $b'_i := 0$
        if received at least $f + 1$ times 1 then
            $b'_i := 1$
        denote by $o'_i$ the output of recursive call on node set $V'$ with
          inputs $b'_i$
        broadcast $o'_i$
    if strong = 0 then
        set $op_i$ to majority value received from nodes in $V'$ (breaking a
          tie arbitrarily)
return $op_i$
```

V'. We conclude that Lemma 10.3 applies to Algorithm 10.2 with the above modification to the statement.

As we have also shown that at least one of the recursive calls succeeds, agreement and validity follow as in Corollaries 10.2 and 10.4, respectively. □

Lemma 10.7. *Algorithm 10.2 terminates in $\mathcal{O}(n)$ rounds.*

Proof. We claim that the total number of (recursive) calls of Algorithm 10.2 is $2n - 1$. This follows from the fact that the resulting binary recursion tree has n leafs (the instances with a single node) and each inner node except for the root has degree 3, i.e., the number of leaves equals the number of inner nodes plus 2.

As, apart from the recursive calls, each instance requires 6 rounds, the claim follows. □

Lemma 10.8. *The total number of bits communicated by the nodes in V_g when executing Algorithm 10.2 is $\mathcal{O}(n^2)$.*

Proof. Denote by $B(n)$ the number of bits communicated in an instance with n nodes. Clearly, in six rounds of communication, at most $6n^2$ bits are sent. Hence, $B(n) \leq 6n^2 + B(n_1) + B(n_2)$, where $n_1 + n_2 = n$. Note that in the algorithm $f \in n/3 + \Theta(1)$ and, in each of the two recursive calls, $|V'| \in 3f/2 + \Theta(1)$. Hence, if n is at least a sufficiently large constant, we have that $\max\{n_1, n_2\} \leq 2n/3$. Otherwise, the remaining number of recursive calls is constant, and we can bound $B(n) \in \mathcal{O}(n^2)$. Therefore, $B(n) \in 2B(2n/3) + \mathcal{O}(n^2)$ for all $n \in \mathbb{N}$. By the master theorem, this implies that $B(n) \in \mathcal{O}(n^2)$. □

Theorem 10.9. *Binary consensus in systems with $f < n/3$ faults can be solved in $\mathcal{O}(f)$ rounds with 1-bit messages, where the total number of communicated bits is $\mathcal{O}(nf)$.*

Proof. For $f \in \Omega(n)$, this is shown for Algorithm 10.2 by Lemmas 10.6, 10.7, and Lemma 10.8. For smaller values of f, run the algorithm on $3f + 1$ nodes, have them broadcast their output, and let each node output the majority value. This adds one round and $\mathcal{O}(fn)$ bits to the previously spent $\mathcal{O}(f)$ rounds and $\mathcal{O}(f^2)$ bits. □

10.3 Running Time Lower Bound

As promised earlier, we prove now that any (deterministic) consensus algorithm must run for at least $f + 1$ rounds in the worst case. In fact, we will show this for a much weaker fault model: crash faults.

Definition 10.10 (Crash Faults)**.** *If node $v \in V$ crashes in round $r \in \mathbb{N}$, it operates like a non-faulty node in rounds $1, \ldots, r - 1$, does nothing at all in rounds $r + 1, r + 2, \ldots$, and in round r sends an arbitrary subset of the messages it would send according to the algorithm.*

Crashing nodes fail in a well-organized fashion. They do not lie, we do not have to care about getting them up to speed again later, and by requiring that nodes always send messages to each other in each round, nodes will learn that a node failed from not receiving a message from the node. None of this affects the worst-case running time lower bound in any way — regardless of whether we consider Byzantine or crash faults, the bound of $f + 1$ rounds turns out to be tight.

We will show this lower bound now by a straightforward inductive argument. The key ingredient is the following definition.

Definition 10.11 (Pivotal Nodes)**.** *Observe that an execution in the synchronous model with crash faults is fully determined by specifying the node inputs and, for each node, whether it crashes and, if so, in which round and which of its messages of this round get sent. Given an execution $\mathcal{E}$ of a consensus algorithm with at most $n - 2$ crash faults and a node $v \in V$ that does not crash in $\mathcal{E}$, we call v* pivotal in round r (of $\mathcal{E}$) *if changing $\mathcal{E}$ by crashing v in round r of $\mathcal{E}$ without v sending any messages results in an execution with different output (the execution does have an output, because at least one node does not crash).*

In order to anchor the induction, we need to show that such nodes exist.

Lemma 10.12. *There is a fault-free execution with a node that is pivotal in round* 1*.*

Proof. Consider executions $\mathcal{E}_i$, $i \in [n+1]$, which are fault-free with node $j \in V$ having input 0 if $j > i$ and input 1 otherwise. By validity, $\mathcal{E}_0$ has output 0 and $\mathcal{E}_1$ has output 1. Thus, there must be some $i \in [n]$ with the property that $\mathcal{E}_i$ has output 0 and $\mathcal{E}_{i+1}$ has output 1. Consider the execution $\mathcal{E}'$ obtained by crashing node $i+1$ in round 1, without $i+1$ getting any messages out. If $\mathcal{E}'$ has output 0, $i+1$ is pivotal in round 1 of execution $\mathcal{E}_{i+1}$; if $\mathcal{E}'$ has output 1, $i+1$ is pivotal in round 1 of execution $\mathcal{E}_i$. □

The induction step works the same way, except that the inputs are replaced by, for each node, the decision whether the pivotal node crashing in round r sends a message to the node or not.

Lemma 10.13. *Suppose* $0 \leq f \leq n-3$ *and* $\mathcal{E}$ *is an execution with* f *failing nodes, one in each round* $1, \dots, f$*, that has a pivotal node in round* $f+1$*. Then there is an execution* $\mathcal{E}'$ *which differs from* $\mathcal{E}$ *only in that this pivotal node crashes in round* $f+1$ *and satisfies that there is a pivotal node in round* $f+2$*.*

Proof. For $i \in [n+1]$, define $\mathcal{E}_i$ by having the pivotal node of $\mathcal{E}$ crash in round $f+1$ and succeed in sending its message for that round to node $j \in \{1, \dots, n\}$ if and only if $j > i$. As we crashed a pivotal node, we know that $\mathcal{E}_0$ and $\mathcal{E}_n$ have different outputs. Thus, there must be some i for which $\mathcal{E}_i$ and $\mathcal{E}_{i+1}$ have different outputs. Now consider the executions $\mathcal{E}_i'$ and $\mathcal{E}_{i+1}'$ obtained from $\mathcal{E}_i$ and $\mathcal{E}_{i+1}$, respectively, in which node $i+1$ crashes in round $f+2$ without sending any messages. The only difference between these executions is whether $i+1$ received the message from the crashing node in round $f+1$ or not; as $i+1$ does not get a message out telling anyone of this difference, the outputs of $\mathcal{E}_i'$ and $\mathcal{E}_{i+1}'$ are the same. Thus, either $\mathcal{E}_i$ and $\mathcal{E}_i'$ have different outputs or $\mathcal{E}_{i+1}$ and $\mathcal{E}_{i+1}'$ have different outputs, i.e., either $i+1$ is pivotal in round $f+2$ of $\mathcal{E}_i$ or it is pivotal in round $i+1$ or $\mathcal{E}_{i+1}$. □

Corollary 10.14. *Any consensus algorithm has an execution with a pivotal node in round* $\min\{f, n-2\}$*.*

Theorem 10.15. *Any consensus algorithm has worst-case running time at least* $\min\{f+1, n-1\}$*.*

Proof. Consider the execution $\mathcal{E}$ with a pivotal node in round $\min\{f, n-2\}$ guaranteed to exist by Corollary 10.14, as well as the execution $\mathcal{E}'$ obtained by crashing the pivotal node in round $\min\{f, n-2\}$. The two executions have different output, but at all nodes but the pivotal one, the only difference to be observed before round $\min\{f+1, n-1\}$ is whether the respective message from the pivotal node in round $\min\{f, n-2\}$ was received or not.

Assume for contradiction that, in both executions, the (at least two) non-crashed nodes terminate by the end of round $\min\{f, n-2\}$. Let $i, j \in V$ be two such nodes crashing in neither $\mathcal{E}$ nor $\mathcal{E}'$. These nodes must also terminate in the execution $\mathcal{E}''$ in which the pivotal node sends its message to i, but does not send its message to j: To i, this execution is indistinguishable from $\mathcal{E}$ before round $\min\{f+1, n-1\}$, and for j it is indistinguishable from $\mathcal{E}$. However, this indistinguishability implies that they also output the same values as in $\mathcal{E}$ and $\mathcal{E}'$, respectively. As these values differ, this violates agreement and hence is a contradiction. We conclude that our assumption must be wrong and there is some execution of the algorithm in which not all nodes terminate before round $\min\{f+1, n-1\}$. □

Remarks:

- The above proof was for binary consensus, but it also works if more than two output values are possible. The salient point is that there are different outputs!

- The same arguments apply even if we restrict the fault model *further.* We could require that, in each round, nodes send their outgoing messages in some specific order; the algorithm could even choose. Thus, a crash would not result in an arbitrary subset of nodes receiving their messages, but rather a prefix of the message sequence being received. Still, the same lower bound applies, with the same proof, where the only difference is that the order in which we list the nodes when defining executions is given by the pivotal nodes' sending sequence for the respective round.

- As mentioned before, randomization can result in faster algorithms.

Bibliographic Notes

Consensus and its variants is a central problem in distributed computing. Thus, any list of references would be no more than a scratch in the tip of the iceberg. Some books addressing the topic are [Lyn96, Ray10]. The Phase King protocol by Berman, Garay, and Perry was introduce in [BGP89]. The recursive version is provided in [BGP92]. The time lower bound was shown by Fischer and Lynch [FL82].

A lower bound of $\Omega(nf)$ on the message complexity is shown in [DR85]. Arguably, this bound is trivial, but the interesting part is that the paper shows a tight bound of $\Theta(n+f^2)$ on the number of messages required with cryptographic signatures. In this case, the total number of bits is still $\Omega(nf)$, but the signatures permit to prove that an output is valid with a single message. As a remark on connectivity requirements, both with and without cryptography node degrees must be larger than f. However, without cryptography, it's not hard to see that the majority of neighbors of each non-faulty nodes must be non-faulty, while with cryptographic signatures, it is sufficient, if the network is still connected when removing faulty nodes. With cryptographic assumptions, it is also sufficient if $f < n/2$. Naturally, all of this requires the ability to perform decoding and encoding operations, cryptographic hardness assumptions (i.e., the adversary can't break protocols by brute force), and that the adversary can't directly obtain information about internal states of correct nodes.

Bibliography

[BGP89] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Towards Optimal Distributed Consensus (Extended Abstract). In *Symposium on Foundations of Computer Science (FOCS)*, pages 410–415, 1989.

[BGP92] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. *Bit Optimal Distributed Consensus*, pages 313–321. Springer US, 1992.

[DR85] Danny Dolev and Rüdiger Reischuk. Bounds on Information Exchange for Byzantine Agreement. *J. ACM*, 32(1):191–204, 1985.

[FL82] Michael J. Fischer and Nancy A. Lynch. A Lower Bound for the Time to Assure Interactive Consistency. *Inf. Process. Lett.*, 14(4):183–186, 1982.

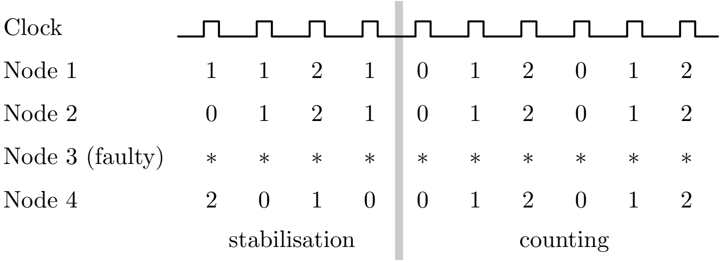
[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[Ray10] Michel Raynal. *Fault-tolerant Agreement in Synchronous Message-passing Systems*. Morgan & Claypool Publishers, 2010.

Lecture 11

Synchronous Counting

Before getting to self-stabilizing pulse synchronization in the next lecture, we consider the related task of *synchronous counting.* In synchronous counting, the goal is to establish a self-stabilizing joint counter (modulo some $2 \leq C \in \mathbb{N}$), despite $f < n/3$ Byzantine faults. This means the good traces are those in which for each round r, it holds for all $v, w \in V_g$ that $c(v,r) = c(w,r)$ and $c(v,r+1) = c(v,r) + 1 \bmod C$.



Clock										
Node 1	1	1	2	1	0	1	2	0	1	2
Node 2	0	1	2	1	0	1	2	0	1	2
Node 3 (faulty)	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$
Node 4	2	0	1	0	0	1	2	0	1	2
	stabilisation				counting					

Despite being instructive for the approach we'll take to pulse synchronization in the next lecture, this is in itself a very useful subroutine. Once the synchronous abstraction is established by a pulse synchronization algorithm, it makes sense to ask for a common numbering of the pulses, allowing for implicit coordination. For instance, this way the nodes can call a subroutine every C rounds without further communication overhead.

11.1 Synchronous Counting vs. Consensus

The first observation is that counting is no easier than (synchronous) consensus.

Lemma 11.1. *A synchronous C-counting algorithm with stabilization time S implies a synchronous C-valued consensus algorithm terminating in S rounds, which satisfies the same bounds on message and bit complexity as the original counting algorithm.*

Proof. Once stabilized, the counting algorithm guarantees a good trace, i.e., the correct nodes will jointly count modulo C. For each $c \in [C]$, denote by $\vec{x}(c)$ the state vector of the correct nodes in some round $r \geq S$ in which the count is

c. Our consensus algorithm now operates as follows. Each $v \in V_g$ runs a local instance of the counting algorithm for S rounds, where given input $c \in [C]$ it initializes its state to $x_v(c)$. At the end of round S, it outputs $c(v, S) - S \bmod C$.

The claims about running time and communication complexity are trivially satisfied, so it remains to show agreement and validity. Agreement is immediate from the fact that the counting algorithm stabilized no later than round S, i.e., $c(v, S) = c(w, S)$ for all $v, w \in V_g$. Concerning validity, observe that the initialization ensures that if each $v \in V_g$ has input $c \in [C]$, then the initial state of the counting algorithm is $\vec{x}(c)$. As this is a system state *after* stabilization, regardless of the behavior of faulty nodes, the correct nodes must increment their counters by exactly 1 modulo C in the following rounds. Thus, in round S, it holds that $c(v, S) = c + S \bmod C$, and each $v \in V_g$ outputs $c + S - S \bmod C = c$. □

The other direction is not as straightforward. However, it is not hard to come up with a reduction that translates running time to stabilization time if we neglect communication.

Lemma 11.2. *Any synchronous C-valued consensus algorithm terminating in R rounds implies a synchrounous C-counting algorithm with stabilization time $\mathcal{O}(R)$.*

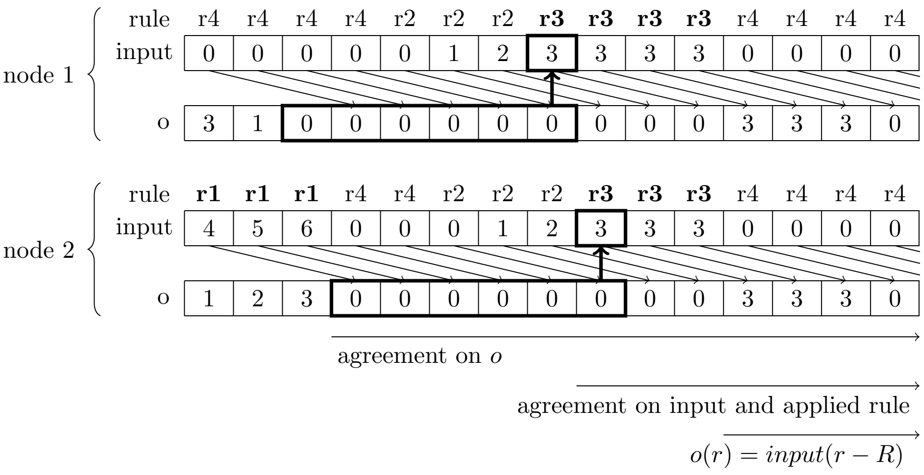
Proof. Given the consensus algorithm, we solve C-counting as follows. In each synchronous round, we start a new consensus instance that will generate an output value $c(v, r + R)$ at each node $v \in V_g$ exactly R rounds later (which will double as node v's counter value); if the consensus instance terminates earlier at v, it will simply store the output value until it is needed. Note that, while we have no guarantees about the outputs in the first R rounds (as initial states are arbitrary), in all rounds $r \geq R$ all correct nodes will output the same value $c(r) = c(v, r)$ (by the agreement property of consensus). Hence, if we define the input value $f(v, r)$ of node $v \in V_g$ as a function of the most recent $\mathcal{O}(R)$ output values at node v, after $2R$ rounds all nodes will start using identical inputs $f(r) = f(v, r)$ and, by validity of the consensus algorithm, reproduce these inputs as output R rounds later (cf. Figure 11.1). In light of these considerations, it is sufficient to determine an input function f from the previous $\mathcal{O}(R)$ outputs to values $[C]$ so that counting starts within $\mathcal{O}(R)$ rounds, assuming that the output of the consensus algorithm in round $r + R$ equals the input determined at the end of round r.

We define the following input function, where all values are taken modulo C:

$$\text{input}(r) := \begin{cases} c + R & \text{if} \quad (o(r - R + 1), \ldots, o(r)) = (c - R + 1, \ldots, c) \\ x + R & \text{if} \quad \begin{array}{l}(o(r - 2R + 1 - x), \ldots, o(r)) = (0, \ldots, 0, 1, \ldots, x) \\ \text{for some } x \in [R]\end{array} \\ x & \text{if} \quad \begin{array}{l}(o(r - R + 1 - x), \ldots, o(r)) = (0, \ldots, 0) \\ \text{for maximal } x \in [R]\end{array} \\ 0 & \text{else.} \end{cases}$$

In the setting discussed above, it is straightforward to verify the following properties of input:

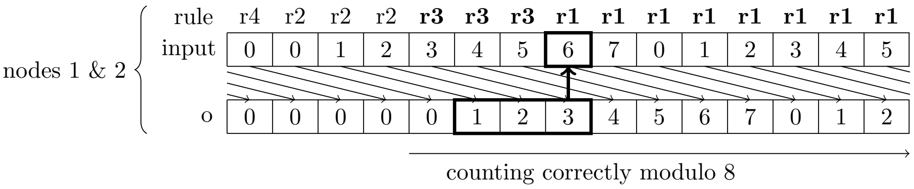
- Always exactly one of the rules applies, i.e., input is well-defined.



node 1	rule	r4	r4	r4	r4	r2	r2	r2	**r3**	**r3**	**r3**	**r3**	r4	r4	r4	r4
	input	0	0	0	0	0	1	2	3	3	3	3	0	0	0	0
	o	3	1	0	0	0	0	0	0	0	0	0	3	3	3	0
node 2	rule	**r1**	**r1**	**r1**	r4	r4	r2	r2	r2	**r3**	**r3**	**r3**	r4	r4	r4	r4
	input	4	5	6	0	0	0	1	2	3	3	3	0	0	0	0
	o	1	2	3	0	0	0	0	0	0	0	0	3	3	3	0

Figure 11.1: Part of an execution of two nodes running the C-counting algorithm given in the proof of Lemma 11.2, for $C = 8$ and $R = 3$. The execution progresses from left to right, each box representing a round. On top of the input field the applied rule (1 to 4) to compute the input is displayed. Displayed are the initial phases of stabilization: (i) after R rounds agreement on the output is guaranteed by consensus, (ii) then agreement on the input and the applied rule is reached, and (iii) another R rounds later the agreed upon outputs are the agreed upon inputs shifted by 3 rounds.

- If the outputs counted modulo C for $2R$ consecutive rounds, they will do so forever (by induction, using the first rule); cf. Figure 11.2.
- If this does not happen within $\mathcal{O}(R)$ rounds, there will be R consecutive rounds where input 0 will be used (by the third and the last rule), cf. Figure 11.2.
- Once R consecutive rounds with input 0 occurred, inputs $1, \ldots, 2R$ will be used in the following $2R$ rounds (by the second and third rule).
- Finally, the algorithm will commence counting correctly (by the first rule).



nodes 1 & 2	rule	r4	r2	r2	r2	**r3**	**r3**	**r3**	**r1**	**r1**	**r1**	**r1**	**r1**	**r1**	**r1**	**r1**
	input	0	0	1	2	3	4	5	6	7	0	1	2	3	4	5
	o	0	0	0	0	0	1	2	3	4	5	6	7	0	1	2

Figure 11.2: Extension of the execution shown in Figure 11.1. Nodes have already agreed upon inputs and outputs so that the latter just reproduce the inputs from R rounds ago. The rules now make sure that the nodes start counting modulo 8 in synchrony, always executing rule 1.

Overall, if each node i computes its input $F_i(r)$ from its local view of the previous outputs using input, the algorithm will start counting correctly within $S \in \mathcal{O}(R)$ rounds. □

Remarks:

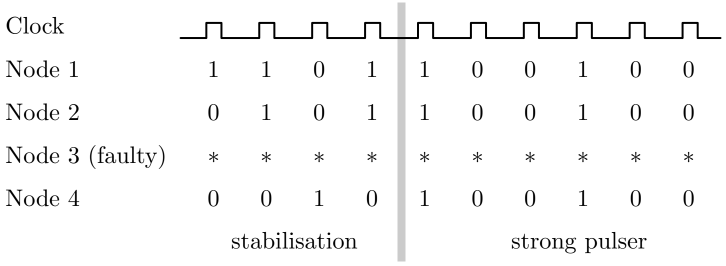
- The second reduction shows that the time complexities of both tasks are, up to a constant factor, identical.
- However, reduction from counting to consensus is inefficient in terms of communication and computation, as there are always R consensus instances running concurrently.
- Resolving this issue will be more challenging, as we can't simply circumvent the issue that the correct nodes don't agree on round numbers any more when using consensus as a subroutine any more by just starting an instance each round.

11.2 Pulsers

As useful tools, we introduce two tasks that are closely related to counting, but not exactly the same. The first one is, essentially, just slightly rephrasing the counting task.

Definition 11.3 (Strong pulser)**.** *An algorithm P is an f-resilient strong Ψ-pulser that stabilizes in $S(P)$ rounds if it satisfies the following conditions in the presence of at most f faulty nodes. Each node $v \in V_g$ produces an output bit $p(v,r) \in \{0,1\}$ on each round $r \in \mathbb{N}$. We say that v generates a pulse in round r if $p(v,r) = 1$ holds. We require that there is a round $r_0 \leq S(P)$ such that:*

1. *For any $v \in V_g$ and round $r = r_0 + k\Psi$, where $k \in \mathbb{N}_0$, it holds that $p(v,r) = 1$.*
2. *For any $v \in V_g$ and round $r \geq r_0$ satisfying $r \neq r_0 + k\Psi$ for $k \in \mathbb{N}_0$, we have $p(v,r) = 0$.*



Clock										
Node 1	1	1	0	1	1	0	0	1	0	0
Node 2	0	1	0	1	1	0	0	1	0	0
Node 3 (faulty)	*	*	*	*	*	*	*	*	*	*
Node 4	0	0	1	0	1	0	0	1	0	0

Figure 11.3: An example execution of a strong 3-pulser on $n = 4$ nodes with $f = 1$ faulty node.

Lemma 11.4. *Let $C \in \mathbb{N}$ and $\Psi \in \mathbb{N}$. If C divides Ψ, then a strong Ψ-pulser that stabilizes in S rounds implies a synchronous C-counter that stabilizes in at most S rounds. If Ψ divides C, then a synchronous C-counter that stabilizes in S rounds implies a strong Ψ-pulser that stabilizes in at most $S + \Psi - 1$ rounds.*

Proof. For the first claim, set $c(v, r) = 0$ in any round r for which $p(v, r) = 1$ and $c(v, r) = c(v, r - 1) + 1 \bmod C$ in all other rounds. For the second claim, set $p(v, r) = 1$ in all rounds r in which $c(v, r) \bmod \Psi = 0$ and $p(v, r) = 0$ in all other rounds. □

Remarks:

- Another way of interpreting this relation is to view a strong Ψ-pulser as a different encoding of the output of a Ψ-counter: since the system is synchronous, it suffices to communicate when the counter overflows to value 0 and otherwise count locally. This saves bandwidth when communicating the state of the counter.
- The additive overhead of Ψ will not matter to us, as we will recursively construct strong pulsers, deriving a counter only in the very end.

A weak Φ-pulser is similar to a strong pulser, but does not guarantee a fixed frequency of pulses. However, it guarantees to *eventually* generate a pulse followed by $\Phi - 1$ rounds of silence.

Definition 11.5 (Weak pulsers)**.** *An algorithm W is an f-resilient weak Φ-pulser that stabilizes in $S(W)$ rounds if the following holds. In each round $r \in \mathbb{N}$, each node $v \in V_g$ produces an output $a(v, r)$. Moreover, there exists a round $r_0 \leq S(W)$ such that*

1. *for all $v, w \in V_g$ and all rounds $r \geq r_0$, $a(v, r) = a(w, r)$,*
2. *$a(v, r_0) = 1$ for all $v \in V_g$, and*
3. *$a(v, r) = 0$ for all $v \in V_g$ and $r \in \{r_0 + 1, \ldots, r_0 + \Phi - 1\}$.*

We say that on round r_0 a good *pulse is generated by W.*

Figure 11.4 illustrates a weak 4-pulser.

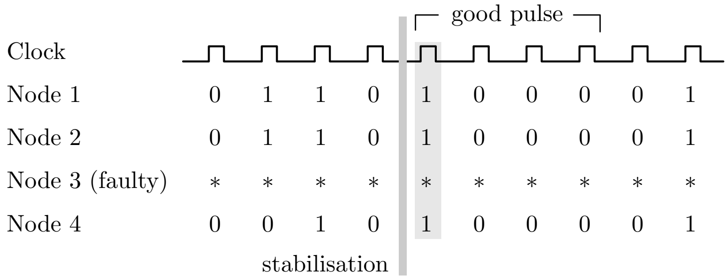
Remarks:

- While the definition formally only asks for one good pulse, the fact that the algorithm guarantees this property for any starting state implies that there is a good pulse at least every $S(W)$ rounds.
- Weak pulsers are (surprise!) easier to construct than strong pulsers. Yet, they are good enough to *eventually* get a consensus instance to be executed correctly, using the good pulse as starting shot for the execution of the consensus algorithm. This we can use to stabilize a strong pulser.

Constructing Strong Pulsers from Weak Pulsers

For constructing a strong Ψ-pulser, we assume that we have the following f-resilient algorithms available:

- an $R(C)$-round Ψ-valued consensus algorithm C and
- a weak Φ-pulser W for some $\Phi \geq R(C)$.



Node 1	0	1	1	0	1	0	0	0	0	1
Node 2	0	1	1	0	1	0	0	0	0	1
Node 3 (faulty)	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$
Node 4	0	0	1	0	1	0	0	0	0	1

Figure 11.4: An example execution of a weak 4-pulser on $n = 4$ nodes with $f = 1$ faulty node. Eventually, a good pulse is generated, which is highlighted. A good pulse is followed by three rounds in which no correct node generates a pulse. In contrast, the pulse two rounds earlier is not good, as it is followed by only one round of silence.

Variables. Beside the variables of the weak pulser W and (a single copy of) C, our construction of a strong Ψ-pulser uses the following local variables:

- $a(v,r) \in \{0,1\}$ is the output variable of the weak Φ-pulser W,
- $b(v,r) \in \{0,1\}$ is the output variable of the strong Ψ-pulser we are constructing,
- $c(v,r) \in [\Psi]$ is the local counter keeping track on when the next pulse occurs, and
- $d(v,r) \in \{1,\dots,R(C)\} \cup \{\perp\}$ keeps track of how many rounds an instance of C has been executed since the last pulse from the weak pulser W. The value $\perp$ denotes that the consensus routine has stopped.

Strong pulser algorithm. The algorithm is as follows. Each node v executes the weak Φ-pulser algorithm W in addition to the following instructions on each round $r \in \mathbb{N}$:

1. If $c(v,r) = 0$, then set $b(v,r) = 1$ and otherwise $b(v,r) = 0$.
2. Set $c'(v,r) = c(v,r)$.
3. If $d(v,r) \neq \perp$, then
   (a) Execute the instructions of C for round $d(v,r)$.
   (b) If $d(v,r) \neq R(C)$, set $d(v,r+1) = d(v,r) + 1$.
   (c) If $d(v,r) = R(C)$, then
      i. Set $c'(v,r) = y(v,r) + R(C) \bmod \Psi$, where $y(v,r)$ is the output value of C.
      ii. Set $d(v,r+1) = \perp$.
4. Update $c(v,r+1) = c'(v,r) + 1 \bmod \Psi$.

5. If $a(v,r)=1$, then
    (a) Start a new instance of C using $c'(v,r)$ as input (resetting all state variables of C).
    (b) Set $d(v,r+1)=1$.

In the above algorithm, the first step simply translates the counter value to the output of the strong pulser. We then use a temporary variable $c'(v,r)$ to hold the counter value, which is overwritten by the output of C (increased by $R(C) \bmod \Psi$) if it completes a run in this round. In either case, the counter value needs to be increased by 1 mod Ψ for the next round. The remaining code does the bookkeeping for an ongoing run of C and starting a new run if the weak pulser generates a pulse.

Observe that in the above algorithm, each node only sends messages related to the weak pulser W and the consensus algorithm C. Thus, there is no additional overhead in communication and the message size is bounded by $M(W)+M(C)$, where $M(\cdot)$ denotes the maximum message size of an algorithm. Hence, it remains to show that the local counters $c(v,r)$ implement a strong Ψ-counter.

Theorem 11.6. *The variables $c(v,r)$ in the above algorithm implement a synchronous Ψ-counter that stabilizes in $S(W)+R(C)+1$ rounds and uses messages of at most $M(W)+M(C)$ bits.*

Proof. Suppose round $r_0 \leq S(W)$ is as in Definition 11.5, that is, $a(v,r)=a(w,r)$ for all $r \geq r_0$, and a good pulse is generated in round r_0. Thus, all correct nodes participate in simulating an instance of C during rounds $r_0+1,\dots,r_0+R(C)$, since no pulse is generated during rounds $r_0+1,\dots,r_0+R(C)-1$, and thus, also no new instance is started in the last step of the code during these rounds.

By the agreement property of the consensus routine, it follows that $c'(v,r_0+R(C))=c'(w,r_0+R(C))$ for all $v,w \in V_g$ after Step 3ci. By Steps 2 and 4, the same will hold for both $c(\cdot,r')$ and $c'(\cdot,r')$, $r'>r_0+R(C)$, provided that we can show that in rounds $r'>r$, Step 3ci never sets $c'(v,r)$ to a value different than $c(v,r)$ for any $v \in V_g$; as this also implies that $c(v,r'+1)=c(v,r')+1 \bmod \Psi$ for all $v \in V_g$ and $r'>r_0+R(C)$, this will complete the proof.

Accordingly, consider any execution of Step 3ci in a round $r'>r_0+R(C)$. The instance of C terminating in this round was started in round $r'-R(C)>t_0$. However, in this round the weak pulser must have generated a pulse, yielding that, in fact, $r'-R(C) \geq r_0+R(C)$. Assuming for contradiction that r' is the earliest round in which the claim is violated, we thus have that $c'(v,r'-R(C))=c'(w,r'-R(C))$ for all $v,w \in V_g$, i.e., all correct nodes used the same input value c for the instance. By the validity property of C, this implies that $v \in V_g$ outputs $y(v,r')=c$ in round r' and sets $c'(v,r')=c+R(C) \bmod \Psi$. However, since r' is the earliest round of violation, we already have that $c'(v,r')=c(v,r')=c+R(C) \bmod \Psi$ after the second step, contradicting the assumption and showing that the execution stabilized in round $r_0+R(C)+1 \leq S(W)+R(C)+1$. □

Together with Lemma 11.4, we get the following corollary.

Corollary 11.7. *Let $\Psi>1$. Suppose that there exists an f-resilient Ψ-value consensus routine C and a weak Φ-pulser W, where $\Phi \geq R(C)$. Then there exists an f-resilient strong Ψ-pulser P that*

- *stabilizes in time* $S(P) \leq R(C) + S(W) + \Psi$*, and*
- *uses messages of size at most* $M(P) \leq M(C) + M(W)$ *bits.*

Remarks:

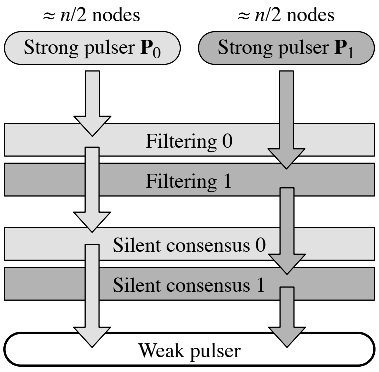
- This straightforward construction reduces our task to designing weak pulsers.
- Even though we "havn't done much," constructing weak pulsers is significantly easier.
- This is an example where the hardest part is to come up with the right question, or rather problem to solve. By giving rise to the questions "can we obtain strong pulsers from weak ones?" and "can we construct weak pulsers?" the notion of weak pulsers breaks the question "can we construct strong pulsers" into (as it turns out) more managable tasks.

11.3 Weak from (less Resilient) Strong Pulsers

Having seen that we can construct strong pulsers from weak pulsers using a consensus algorithm, the missing piece is the existence of efficient weak pulsers. We now devise a recursive construction of a weak pulser from strong pulsers of smaller resilience. Given that a 0-resilient pulser is trivial and that we can obtain strong pulsers from weak ones without losing resilience, this is sufficient for constructing strong pulsers of optimal resilience from consensus algorithms of optimal resilience.

At a high level, we take the following approach (see Figure 11.5):

1. Partition the network into two parts, each running a strong pulser (with small resilience). Our construction guarantees that at least one of the strong pulsers stabilizes.
2. Filtering of pulses generated by the strong pulsers:
   a) Nodes consider the observed pulses generated by the strong pulsers as *potential* pulses.
   b) Since one of the strong pulsers may not stabilize, it may generate *spurious* pulses, that is, pulses that only a subset of the correct nodes observe.
   c) We limit the *frequency* of the spurious pulses using a filtering mechanism based on threshold voting.
3. We force any spurious pulse to be observed by either all or none of correct nodes by employing a *silent consensus* routine. In silent consensus, no message is sent (by correct nodes) if all correct nodes have input 0. Thus, if all nodes actually participating in an instance have input 0, non-participating nodes behave *as if they participated* with input 0. This avoids the chicken-and-egg problem of having to solve consensus on participation in the consensus routine. We make sure that if any node uses input 1, i.e., the consensus routine may output 1, all nodes participate. Thus, when a pulse is generated, all correct nodes agree on this.



1. The network is divided into two blocks. Each block runs a strong pulser instance, where the pulsers have coprime frequencies.

2. All n nodes participate in filtering spurious pulses.

3. Use consensus to agree whether the block generated a pulse recently.

4. A pulse is generated if one of the consensus instances outputs “1”.

Figure 11.5: Overview of the weak pulser construction. Light and dark grey boxes correspond to steps of block 0 and 1, respectively. The small rounded boxes denote the pulser algorithms P_i that are run (in parallel) on two disjoint sets of roughly $n/2$ nodes, whereas the wide rectangular boxes denote to the filtering steps in which all of the n nodes are employed. The arrows indicate the flow of information for each block.

4. If a potential pulse generated by one of the pulsers both passes the filtering step and the consensus instance outputs “1”, then a weak pulse is generated.

The Filtering Construction

Our goal is to construct a weak Φ-pulser (for sufficiently large Φ) with resilience f. We partition the set of n nodes into two disjoint sets V_0 and V_1 with n_0 and n_1 nodes, respectively. Thus, we have $n = n_0 + n_1$. For $i \in \{0, 1\}$, let P_i be an f_i-resilient strong Ψ_i-pulser. That is, P_i generates a pulse every Ψ_i rounds once stabilized, granted that V_i contains at most f_i faulty nodes. Nodes in block i execute the algorithm P_i. Our construction tolerates $f = f_0 + f_1 + 1$ faulty nodes. Since we consider Byzantine faults, we require the additional constraint that $f < n/3$.

Let $a_i(v, r) \in \{0, 1\}$ indicate the output bit of P_i for a node $v \in V_i$. Note that we might have a block $i \in \{0, 1\}$ that contains more than f_i faulty nodes. Thus, it is possible that the algorithm P_i never stabilizes. In particular, we might have the situation that some of the nodes in block i produce a pulse, but others do not. We say that a pulse generated by such a P_i is *spurious*. We proceed by showing how to filter out such spurious pulses if they occur too often.

Filtering rules. We define five variables with the following semantics:

- $m_i(v, r{+}1)$ indicates whether at least $n_i - f_i$ nodes $u \in V_i$ sent $a_i(u, r) = 1$,
- $M_i(v, r{+}1)$ indicates whether at least $n - f$ nodes $u \in V$ sent $m_i(u, r) = 1$,

- $\ell_i(v,r)$ indicates when was the last time block i triggered a (possibly spurious) pulse,
- $x_i(v,r)$ is a *cooldown counter* that indicates how many rounds any firing events coming from block i are ignored, and
- $b_i(v,r)$ indicates whether node v accepts a firing event from block i.

The first two of the above variables are set according to the following rules:

- $m_i(v,r+1) = 1$ if and only if $|\{w \in V_i : a_i(v,w,r) = 1|\} \geq n_i - f_i$,
- $M_i(v,r+1) = 1$ if and only if $|\{w \in V : m_i(v,w,r) = 1\} \geq n - f$,

where $a_i(v,w,r)$ and $m_i(v,w,r)$ denote the values for $a(\cdot)$ and $m(\cdot)$ node v received from u at the end of round r, respectively. Furthermore, we update the $\ell(\cdot,\cdot)$ variables using the rule

$$\ell_i(v,r+1) = \begin{cases} 0 & \text{if } |\{w \in V : m_i(w,r) = 1\}| \geq f+1, \\ y & \text{otherwise,} \end{cases}$$

where $y = \min\{\Psi_i, \ell_i(v,r)+1\}$ (and, of course, each node $v \in V_g$ performs the update according to the count it perceives). In words, the counter is reset on round $r+1$ if v has proof that at least one correct node w had $m_i(w,r) = 1$, that is, some $w \in V_g$ observed P_i generating a (possibly spurious) pulse.

We reset the cooldown counter x_i whenever suspicious activity occurs. The idea is that it is reset to its maximum value C by node v in the following two cases:

- some other correct node $u \neq v$ observed block i generating a pulse, but the node v did not, or
- block i generated a pulse, but this happened either too soon or too late.

To capture this behaviour, the cooldown counter is set with the rule

$$x_i(v,r+1) = \begin{cases} C & \text{if } M_i(v,r+1) = 0 \text{ and } \ell_i(v,r+1) = 0, \\ C & \text{if } M_i(v,r+1) = 1 \text{ and } \ell_i(v,t) \neq \Psi_i - 1, \\ y & \text{otherwise,} \end{cases}$$

where $y = \max\{x_i(v,r)-1, 0\}$ and $C = \max\{\Psi_0, \Psi_1\} + \Phi + 2$. Finally, a node v accepts a pulse generated by block i if the node's cooldown counter is zero and it saw at least $n-f$ nodes supporting the pulse. The variable $b_i(v,r)$ indicates whether node v accepted a pulse from block i on round r. The variable is set using the rule

$$b_i(v,t) = \begin{cases} 1 & \text{if } x_i(v,r) = 0 \text{ and } M_i(v,r) = 1, \\ 0 & \text{otherwise.} \end{cases}$$

Analysis of the Filtering Construction

We now analyse when the nodes accept firing events generated by the blocks. We say that a block i is correct if it contains at most f_i faulty nodes. Note that since there are at most $f = f_0 + f_1 + 1$ faulty nodes, at least one block $i \in \{0, 1\}$ will be correct. Thus, eventually the algorithm P_i run by a correct block i will stabilize. This yields the following lemma.

Lemma 11.8. *For some $i \in \{0, 1\}$, the strong pulser algorithm P_i stabilizes by round $S(P_i)$.*

We proceed by establishing some bounds on when (possibly spurious) pulses generated by block i are accepted. We start with the case of having a correct block i.

Lemma 11.9. *If block i is correct, then there exists a round $t_0 \leq S(P_i) + 2$ such that for each $v \in V_g$, $M_i(v, r) = 1$ if and only if $r = t_0 + k\Psi_i$ for $k \in \mathbb{N}_0$.*

Proof. If block i is correct, then the algorithm P_i stabilizes by round $S(P_i)$. Hence, there is some $r_0 \leq S(P)$ so that the output variable $a_i(\cdot)$ of P_i satisfies

$$a_i(v, r) = 1 \text{ if and only if } r = r_0 + k\Psi_i \text{ for } k \in \mathbb{N}_0$$

holds for all $r \geq r_0$. We will now argue that $t_0 = r_0 + 2$ satisfies the claim of the lemma.

If P_i generates a pulse on round $r \geq r_0$, then at least $n_i - f_i$ correct nodes $u \in V_i \cap V_g$ have $a_i(u, r) = 1$. Therefore, for all $v \in V_g$ we have $m_i(v, r+1) = 1$, and consequently, $M_i(v, r+2) = 1$. Since block i is correct, there are at most f_i faulty nodes in the set V_i. Observe that by Lemma 11.4 strong pulsers solve synchronous counting, which in turn is as hard as consensus (by Lemma 11.1). This implies that we must have $f_i < n_i/3$, as P_i is a strong f_i-resilient pulser for n_i nodes. Therefore, if P_i does not generate a pulse on round $r \geq r_0$, then at most $f_i < n_i - f_i$ faulty nodes w may claim $a_i(w, t) = 1$. This yields that $m_i(v, t+1) = M_i(v, t+2) = 0$ for all $v \in V_g$. □

We can now establish that a correct node accepts a pulse generated by a correct block i exactly every Ψ_i rounds.

Lemma 11.10. *If block i is correct, then there exists a round $t_1 \leq S(P_i) + 2C$ such that for each $v \in V_g$, $b_i(v, r) = 1$ for any $r \geq r_0$ if and only if $r = t_1 + k\Psi_i$ for $k \in \mathbb{N}_0$.*

Proof. Lemma 11.9 implies that there exists $t_0 \leq S(P_i) + 2$ such that both $M_i(v, t) = 1$ and $\ell_i(v, r) = 0$ hold for $r \geq t_0$ if and only if $r = t_0 + k\Psi_i$ for $k \in \mathbb{N}_0$. It follows that $x_i(v, r+1) = \max\{x_i(v, r) - 1, 0\}$ for all such r and hence $x_i(v, r') = 0$ for all $r' \geq t_0 + C + 2$. The claim now follows from the definition of $b_i(v, r')$, the choice of t_0, and the fact that $\Psi_i \leq C - 2$. □

It remains to deal with the faulty block. If we have Byzantine nodes, then a block i with more than f_i faulty nodes may attempt to generate spurious pulses. However, the filtering mechanism prevents the spurious pulses from occuring too frequently.

Lemma 11.11. *Let $v, v' \in V_g$ and $t > 2$. Suppose that $b_i(v, r) = 1$ and $r' > r$ is minimal such that $b_i(v', r') = 1$. Then $r' = r + \Psi_i$ or $r' > r + C$.*

Proof. Suppose that $b_i(v,r) = 1$ for some correct node $v \in V_g$ and $r > 2$. Since $b_i(v,r) = 1$, $x_i(v,r) = 0$ and $M_i(v,r) = 1$. Because $M_i(v,r) = 1$, there must be at least $n - 2f > f$ correct nodes w such that $m_i(w, r-1) = 1$. Hence, $\ell_i(w,t) = 0$ for every node $w \in V_g$.

Recall that $r' > r$ is minimal so that $b_i(v', r') = 1$. Again, $x_i(v', r') = 0$ and $M_i(v', r') = 1$. Moreover, since $\ell_i(v', r) = 0$, we must have $\ell_i(v', r) < \Psi_i - 1$ for all $r \leq t < r + \Psi_i - 1$. This implies that $r' \geq r + \Psi_i$, as $x_i(v', r') = 0$ and $M_i(v', r') = 1$ necessitate that $\ell_i(v', r' - 1) = \Psi_i - 1$. In the event that $r' \neq r + \Psi_i$, the cooldown counter must have been reset at least once, i.e., $x_i(v', t) = C$ holds for some $r < t \leq r' - C$, implying that $r' > r + C$. □

Remarks:

- The bottomline: The filtering mechanism does not interfere with the output of correct blocks, but it restricts the possible confusion arising from faulty blocks to either sticking to a fixed frequency or being eliminated completely for long enoug (i.e., C rounds).

Using Silent Consensus to Prune Spurious Pulses

The above filtering mechanism prevents spurious pulses from occurring too often: if some node accepts a pulse from block i, then no node accepts a pulse from this block for at least Ψ_i rounds. We now strengthen the construction to enforce that any (possibly spurious) pulse generated by block i will be accepted by either all or none of the correct nodes. In order to achieve this, we employ *silent consensus.*

Definition 11.12 (Silent consensus)**.** *We call a consensus protocol* silent*, if in each execution in which all correct nodes have input* 0*, correct nodes send no messages.*

The idea is that this enables to have consistent executions even if not all correct nodes actually take part in an execution, provided we can ensure that in this case all participating correct nodes use input 0: the non-participating nodes send no messages either, which is the exact same behavior participating nodes would exhibit.

Theorem 11.13. *Any consensus protocol C can be transformed into a silent binary consensus protocol C' with $R(C') = R(C) + 2$ and the same resilience and message size.*

Proof. Exercise. □

For example, plugging in the Phase King protocol, we get the following corollary.

Corollary 11.14. *For any $f < n/3$, there exists a deterministic f-resilient silent binary consensus protocol C with $R(C) \in \Theta(f)$ and $M(C) \in \mathcal{O}(1)$.*

As the filtering construction bounds the frequency at which spurious pulses may occur from above, we can make sure that at each time, only one consensus instance can be executed for each block. However, we need to further preprocess the inputs, in order to make sure that (i) all correct nodes participate in an instance or (ii) no participating correct node has input 1; here, output 1 means agreement on a pulse being triggered, while output 0 results in no action.

Recall that $b_i(v,r) \in \{0,1\}$ indicates whether v observed a (filtered) pulse of the strong pulser P_i in round r. Moreover, assume that C is a silent consensus protocol running in $R(C)$ rounds. We use two copies C_i, where $i \in \{0,1\}$, of the consensus routine C. We require that $\Psi_i \geq R(C)$, which guarantees by Lemm 11.11 that (after stabilization) every instance of C has sufficient time to complete. Adding one more level of voting to clean up the inputs, we arrive at the following routine.

The pruning algorithm. Besides the local variables of C_i, the algorithm will use the following variables for each $v \in V_g$ and round $r \in \mathbb{N}$:

- $y_i(v,r) \in \{0,1\}$ denotes the output value of consensus routine C_i,
- $t_i(v,r) \in \{1,\dots,R(C)\} \cup \{\perp\}$ is a (local) round counter for controlling C_i, and
- $B_i(v,r) \in \{0,1\}$ is the output of block i.

Now each node v executes the following on round r:

1. Broadcast the value $b_i(v,r)$.
2. If $b_i(v,w,r-1) = 1$ for at least $n-2f$ nodes $w \in V$, then reset $t_i(v,r) = 1$.
3. If $t_i(v,r) = 1$, then
   (a) start a new instance of C_i, that is, re-initialise the variables of C_i correctly,
   (b) use input 1 if $b_i(v,w,r-1) = 1$ for at least $n-f$ nodes $w \in V$ and 0 otherwise.
4. If $t_i(v,r) = R(C)$, then
   (a) execute round $R(C)$ of C_i,
   (b) set $t_i(v,r+1) = \perp$,
   (c) set $B_i(v,r+1) = y_i(v,r)$, where $y_i(v,r) \in \{0,1\}$ is the output variable of C_i.

   Otherwise, set $B_i(v,r+1) = 0$.
5. If $t_i(v,r) \notin \{R(C),\perp\}$, then
   (a) execute round $t_i(v,r)$ of C_i, and
   (b) set $t_i(v,r+1) = t_i(v,r) + 1$.

Analysis. Besides the communication used for computing the values $b_i(\cdot)$, the above algorithm uses messages of size $M(C)+1$, as $M(C)$ bits are used when executing C_i and one bit is used to communicate the value of $b_i(v,r)$.

We say that $v \in V_g$ *executes the round* $t \in \{1,\dots,T(C)\}$ of C_i in round r iff $t_i(v,r)=t$. By Lemm 11.11, in rounds $t > R(C)+2$, there is always at most one instance of C_i being executed, and if so, consistently.

Corollary 11.15. *Suppose that $v \in V_g$ executes round 1 of C_i in some round $r > T(C)+2$. Then there is a subset $U \subseteq V_g$ such that each $w \in U$ executes round $t \in \{1,\dots,R(C)\}$ of C_i in round $r+t-1$ and no $u \in V_g \setminus U$ executes any round of C_i in round $r+t-1$.*

Exploiting silence of C_i and the choice of inputs, we can ensure that the case $U \neq V_g$ causes no trouble.

Lemma 11.16. *Let $r > T(C)+2$ and U be as in Corollary 11.15. Then $U = V_g$ or each $w \in U$ has input 0 for the respective instance of C_i.*

Proof. Suppose that $v \in U$ starts an instance with input 1 in round $r' \in \{r - T(C)-1,\dots,r\}$. Then $b_i(w,r'-1)=1$ for at least $n-2f$ nodes $w \in V_g$, since v received $b_i(v,w,r'-1)=1$ from $n-f$ nodes $w \in V$. Thus, each $v' \in V_g$ received $b_i(v',w,r'-1)=1$ from at least $n-2f$ nodes v' and sets $r_i(v',r')=1$, i.e., $U=V_g$. The lemma now follows from Corollary 11.15. □

Recall that if all nodes executing C_i have input 0, non-participating correct nodes behave exactly as if they executed C_i as well, i.e., they send no messages. Hence, if $U \neq V_g$, all nodes executing the algorithm will compute output 0. Therefore, Corollary 11.15, Lemm 11.11, and Lemm 11.16 imply the following corollary.

Corollary 11.17. *In rounds $r > T(C)+2$ it holds that $B_i(v,r)=B_i(w,t)$ for all $v,w \in V_g$ and $i \in \{0,1\}$. Furthermore, if $B_i(v,r)=1$ for $v \in V_g$ and $r > T(C)+2$, then the minimal $r' > r$ so that $B_i(v,r')=1$ (if it exists) satisfies either $r' = r+\Psi_i$ or $r' > r+C = t+\max\{\Psi_0,\Psi_1\}+\Phi+2$.*

Finally, we observe that our approach does not filter out pulses from correct blocks.

Lemma 11.18. *If block i is correct, there is a round $t_2 \leq S(P_i)+2C+R(C)+1$ so that for any $r \geq t_2$, $B_i(v,r)=1$ if and only if $r = t_2 + k\Psi_i$ for some $k \in \mathbb{N}_0$.*

Proof. Lemm 11.10 states the same for the variables $b_i(v,r)$ and a round $t_1 \leq S(P_i)+2C$. If $b_i(v,r)=1$ for all $v \in V_g$ and some round r, all correct nodes start executing an instance of C_i with input 1 in round $r+1$. As, by Corollary 11.15, this instance executes correctly and, by validity of C_i, outputs 1 in round $r+R(C)$, all correct nodes satisfy $B_i(v,r+R(C)+1)=1$. Similarly, $B_i(v,r+R(C)+1)=0$ for such v and any $r \geq t_1$ with $b_i(v,r)=0$. □

Remarks:

- The bottomline: we used consensus to enforce consistency of the outputs of correct nodes.
- In order to resolve the issue that not always all correct nodes will know to participate, we used silent consensus. If *anyone* is set on using input 1 (everything seems to be fine), all correct nodes participate. Otherwise, the participating nodes have input 0, and because the non-participating nodes do not send messages, the fact that the consensus routine is silent means that the run behaves just as if everyone participated with input 0.
- Note how this is similar to how we made the Phase King algorithm work: either the Phase King figures out that someone is stuck with value $b \in \{0,1\}$ and broadcasts b, or no correct node is stuck with a fixed value, so it doesn't matter which value the king broadcasts.

Obtaining the Weak Pulser

Finally, we define the output variable of our weak pulser as

$$B(v,r) = \max\{B_0(v,r), B_1(v,r)\}.$$

As we have eliminated the possibility that $B_i(v,r) \neq B_i(w,r)$ for $v,w \in V_g$ and $r > R(C)+2$, Property W1 holds. Since there is at least one correct block i by Lemma 11.8, Lemma 11.18 shows that there will be good pulses (satisfying Properties W2 and W3) regularly, unless block $1-i$ interferes by generating pulses violating Property W3 (i.e., in too short order after a pulse generated by block i). Here the filtering mechanism comes to the rescue: as we made sure that pulses are either generated at the chosen frequency Ψ_i or a long period of C rounds of generating no pulse is enforced (Corollary 11.17), it is sufficient to choose Ψ_0 and Ψ_1 as coprime multiples of Φ.

Accordingly, we pick $\Psi_0 = 2\Phi$ and $\Psi_1 = 3\Phi$ and observe that this results in a good pulse within $\mathcal{O}(\Phi)$ rounds after the B_i stabilized.

Lemma 11.19. *In the construction described in the previous two subsections, choose $\Psi_0 = 2\Phi$ and $\Psi_1 = 3\Phi$ for any $\Phi \geq R(C)$. Then $B(v,r)$ is the output variable of a weak Φ-pulser with stabilization time* $\max\{S(P_0), S(P_1)\} + \mathcal{O}(\Phi)$.

Proof. We have that $C = \max\{\Psi_0, \Psi_1\} + \Phi + 2 \in \mathcal{O}(\Phi)$. By the above observations, there is a round

$$\begin{aligned} r &\in \max\{S(P_0), S(P_1)\} + R(C) + \mathcal{O}(\Phi) \\ &\subseteq \max\{S(P_0), S(P_1)\} + \mathcal{O}(\Phi) \end{aligned}$$

satisfying the following four properties. For either block $i \in \{0,1\}$, we have by Corollary 11.17 that

1. $B_i(v,r') = B_i(w,r')$ and $B(v,r') = B(w,r')$ for any $v,w \in V_g$ and $r' \geq r$.

Moreover, for a correct block i and for all $v \in V_g$ we have from Lemma 11.18 that

2. $B_i(v,r) = B_i(v,r+\Psi_i) = 1$,

3. $B_i(v,r') = 0$ for all $r' \in \{r+1,\dots,r+\Phi-1\} \cup \{r+\Psi_i+1,\dots,r+\Psi_i+\Phi-1\}$,

and for a (possibly faulty) block $1-i$ we have from Corollary 11.17 that

4. if $B_{1-i}(v,r') = 1$ for some $v \in V_g$ and $r' \in \{r+1,\dots,r+\Psi_i+\Phi-1\}$, then $B_{1-i}(w,r'') = 0$ for all $w \in V_g$ and $r'' \in \{r'+1,\dots,r'+C\}$ that do not satisfy $r'' = r' + k\Psi_{1-i}$ for some $k \in \mathbb{N}_0$.

Now it remains to argue that a good pulse is generated. Suppose that i is a correct block given by Lemma 11.8. By the first property, it suffices to show that a good pulse occurs in round r or in round $r + \Psi_i$. From the second property, we get for all $v \in V_g$ that $B(v,r) = 1$ and $B(v, r+\Psi_i) = 1$. If the pulse in round r is good, the claim holds. Hence, assume that there is a round $r' \in \{r+1,\dots,r+\Psi_i-1\}$ in which another pulse occurs, that is, $B(v,r') = 1$ for some $v \in V_g$. This entails that $B_{1-i}(v,r') = 1$ by the third property. We claim that in this case the pulse in round $r+\Psi_i$ is good. To show this, we exploit the fourth property. Recall that $C > \Psi_i + \Phi$, i.e., $r' + C > r + \Psi_i + \Phi$. We distinguish two cases:

- In the case $i = 0$, we have that $r' + \Psi_{1-i} = r' + 3\Phi = r' + \Psi_0 + \Psi > r + \Psi_0 + \Phi$, that is, the pulse in round $r + \Psi_0 = r + \Psi_i$ is good.
- In the case $i = 1$, we have that $r' + \Psi_{1-i} = r' + 2\Phi < r + 3\Phi = r + \Psi_1$ and $r' + 2\Psi_{1-i} = r' + 4\Phi = r' + \Psi_1 + \Phi > r + \Psi_1 + \Phi$, that is, the pulse in round $r + \Psi_1 = r + \Psi_i$ is good.

In either case, a good pulse occurs by round

$$r + \max\{\Psi_0, \Psi_1\} \in \max\{S(P_0), S(P_1)\} + \mathcal{O}(\Phi). \qquad \square$$

From the above lemma and the constructions discussed in this section, we get the following theorem.

Theorem 11.20. *Let $n = n_0 + n_1$ and $f = f_0 + f_1 + 1$, where $n > 3f$. Suppose that C is an f-resilient consensus algorithm on n nodes and let $\Phi \geq R(C)+2$. If there exist f_i-resilient strong Ψ_i-pulser algorithms on n_i nodes, where $\Psi_0 = 2\Phi$ and $\Psi_1 = 3\Phi$, then there exists an f-resilient weak Φ-pulser W on n nodes that satisfies*

- $S(W) \in \max\{S(P_0), S(P_1)\} + \mathcal{O}(\Phi)$,
- $M(W) \in \max\{M(P_0), M(P_1)\} + \mathcal{O}(M(C))$.

Proof. By Theorem 11.13, we can transform C into a silent consensus protocol C', at the cost of increasing its round complexity by 2. Using C' in the construction, Lemma 11.19 shows that we obtain a weak Φ-pulser with the stated stabilization time, which by construction tolerates f faults. Concerning the message size, note that we run P_0 and P_1 on disjoint node sets. Apart from sending at most $\max\{M(P_0), M(P_1)\}$ bits per round for its respective strong pulser, each node may send up to $M(C) \geq 1$ bits each to each other node for the two copies C_i of C it runs in parallel, plus a constant number of additional bits for the filtering construction including its outputs $b_i(\cdot,\cdot)$. $\square$

Remarks:

- The work is done, we merely need to chain the constructions for weak and strong pulsers recursively now.

11.4 Plugging it Together

Finally, in this section we put the developed machinery to use. As our main result, we show how to recursively construct strong pulsers out of consensus algorithms.

Theorem 11.21. *Suppose that we are given a family of f-resilient consensus algorithms $C(f)$ running on any number $n > 3f$ of nodes in $R(C(f))$ rounds using $M(C(f))$-bit messages, where both $R(C(f))$ and $M(C(f))$ are non-decreasing in f. Then, for any $\Psi \in \mathbb{N}$, $f \in \mathbb{N}_0$, and $n > 3f$, there exists a strong Ψ-pulser P on n nodes that stabilizes in time*

$$S(P) \in (1+o(1))\Psi + \mathcal{O}\left(\sum_{j=0}^{\lceil \log f \rceil} R(C(2^j))\right)$$

and uses messages of size at most

$$M(P) \in \mathcal{O}\left(1 + \sum_{j=0}^{\lceil \log f \rceil} M(C(2^j))\right)$$

bits, where the sums are empty for $f = 0$.

Proof. We show by induction on k that f-resilient strong Ψ-pulsers $P(f, \Psi)$ on $n > 3f$ nodes with the stated complexity exist for any $f < 2^k$, with the addition that the (bounds on) stabilization time and message size of our pulsers are non-decreasing in f. We anchor the induction at $k = 0$, i.e., $f = 0$, for which, trivially, a 0-resilient strong Ψ-pulser with $n \in \mathbb{N}$ nodes is given by one node generating pulses locally and informing the other nodes when to do so. This requires 1-bit messages and stabilizes in $\Psi + 1$ rounds.

Now assume that $2^k \leq f < 2^{k+1}$ for $k \in \mathbb{N}_0$ and the claim holds for all $0 \leq f' < 2^k$. Since $2 \cdot (2^k - 1) + 1 = 2^{k+1} - 1$, there are $f_0, f_1 < 2^k$ such that $f = f_0 + f_1 + 1$. Moreover, as $n > 3f > 3f_0 + 3f_1$, we can pick $n_i > 3f_i$ for both $i \in \{0, 1\}$ satisfying $n = n_0 + n_1$. Let $P(f', \Psi')$ denote a strong Ψ'-pulser that exists by the induction hypothesis for $f' < 2^k$.

We intend to use Ψ-valued consensus algorithm C' on n nodes resilient to f faults that we obtain from $C(f)$ as in Task 1 of Exercise 9. In order to make use of it, we need a weak Φ-pulser, where $\Phi \in \mathcal{O}(\log \Psi) + R(C(f))$ matches the time complexity of C'. Without loss of generality, we may assume that the $\mathcal{O}(\log \Psi)$ term is at least 2, that is, $\Phi \geq 2 + R(C(f))$. We apply Theorem 11.20 to $C(f)$ and $P_i = P(f_i, \Psi_i)$, where $\Psi_0 = 2\Phi$ and $\Psi_1 = 3\Phi$, to obtain a weak Φ-pulser W on n nodes with resilience f, stabilization time of

$$S(W) \in \max\{S(P_0), S(P_1)\} + \mathcal{O}(\Phi)\,,$$

and message size of

$$M(W) \in \max\{M(P_0), M(P_1)\} + \mathcal{O}(M(C(f)))\,.$$

Recall from Task 1 of Exercise 9 that C' uses messages of size $M(C(f))$ bits and runs in $R(C') \leq \Phi$ rounds. We feed the weak pulser W and the multivalued consensus protocol C' into Corollary 11.7 to obtain an f-resilient strong Ψ-pulser P that stabilizes in

$$\begin{aligned} S(P) &\le R(C') + S(W) + \Psi \le S(W) + \Psi + \Phi \\ &\in \max\{S(P_0), S(P_1)\} + \Psi + \mathcal{O}(\Phi) \end{aligned}$$

rounds and has message size bounded by

$$\begin{aligned} M(P) &\le M(W) + M(C(f)) \\ &\in \max\{M(P_0), M(P_1)\} + \mathcal{O}(M(C(f))). \end{aligned}$$

Applying the bounds given by the induction hypothesis to P_0 and P_1, the definitions of Φ, Ψ_0 and Ψ_1, and the fact that both $R(C(f))$ and $M(C(f))$ are non-decreasing in f, we get that the stabilization time satisfies

$$\begin{aligned} S(P) &\in \max\{S(P(f_0, \Psi_0)), S(P(f_1, \Psi_1))\} + \Psi + \mathcal{O}(\Phi) \\ &\subseteq (1 + o(1)) \cdot 3\Phi + \mathcal{O}\left(\sum_{j=0}^{\lceil \log 2^k \rceil} R(C(2^j)) \right) \\ &+ \Psi + \mathcal{O}(\Phi) \\ &\subseteq \Psi + \mathcal{O}(\log \Psi) + \mathcal{O}\left(\sum_{j=0}^{\lceil \log 2^k \rceil} R(C(2^j)) \right) \\ &+ \mathcal{O}(R(C(f))) \\ &\subseteq (1 + o(1))\Psi + \mathcal{O}\left(\sum_{j=0}^{\lceil \log f \rceil} R(C(2^j)) \right), \end{aligned}$$

and message size is bounded by

$$\begin{aligned} M(P) &\in \max\{M(P(f_0, \Psi_0)), M(P(f_1, \Psi_1))\} \\ &\quad + \mathcal{O}(M(C(f))) \\ &\subseteq \mathcal{O}\left(1 + \sum_{j=0}^{\lceil \log 2^k \rceil} M(C(2^j)) \right) + \mathcal{O}(M(C(f))) \\ &\subseteq \mathcal{O}\left(1 + \sum_{j=0}^{\lceil \log f \rceil} M(C(2^j)) \right). \end{aligned}$$

Because we bounded complexities using $\max_i\{S(P_i)\}$, $\max_i\{M(P_i)\}$, $R(C(f))$ and $M(C(f))$, all of which are non-decreasing in f by assumption, we also maintain that the new bounds on stabilization time and message size are non-decreasing in f. Thus, the induction step succeeds and the proof is complete. □

Plugging in the Phase King protocol, we can extract a strong pulser that is optimally resilient, has asymptotically optimal stabilization time, and message size $\mathcal{O}(\log f)$.

Corollary 11.22. *For any $\Psi, f \in \mathbb{N}$ and $n > 3f$, an f-resilient strong Ψ-pulser on n nodes with stabilization time $(1+o(1))\Psi + \mathcal{O}(f)$ and message size $\mathcal{O}(\log f)$ exists.*

Corollary 11.23. *For any $C, f \in \mathbb{N}$ and $n > 3f$, an f-resilient C-counter on n nodes with stabilization time $\mathcal{O}(f + \log C)$ and message size $\mathcal{O}(\log f)$ exists.*

Proof. In the last step of the construction of Theorem 11.21, we do not use Corollary 11.7 to extract a strong pulser, but directly obtain a counter using Theorem 11.6. This avoids the overhead of Ψ due to waiting for the next pulse. Recalling that the $o(\Psi)$ term in the complexity comes from the $\mathcal{O}(\log \Psi)$ additive overhead in time of the multi-value consensus routine, the claim follows. □

Remarks:

- The construction may look awfully complicated, but this is not the result of a high difficulty of the proof.
- Taking into account that the idea that recursion might help is borrowed from the recursive variant of the Phase King protocol, the main challenges were coming up with the idea to break the problem up into the subtasks of constructing weak and strong pulsers, and seeing that silent consensus can be used to circumvent the need for running consensus on whether to run consensus.
- The entire construction works, without any changes, with randomized consensus routines (satisfying certain constraints, which can as easily and generically be achieved as silence). In particular, this yields solutions with stabilization time $\log^{\mathcal{O}(1)} f$.
- Needless to say that you're not expected to know the full details of the construction by heart!

Bibliographic Notes

The synchronous counting problem was dubbed by Dolev and Hoch under the name of *self-stabilizing Byzantine digital clock synchronization* [HDD06]. They provide a linear-time solution based on consensus. The construction given in Lemma 11.2 is a simplification given in a later survey [DFL$^+$15]. The term "synchronous counting" came up later, because "self-stabilizing Byzantine digital clock synchronization" just takes way too long to say (try it out 10 times).

However, (another) Dolev and Welch were the ones who originally introduced the task, in the same article in which they introduce and solve self-stabilizing pulse synchronization [DW04]. They devise an exponential-time solution for counting, which they then adapt to yield an exponential-time self-stabilizing pulse synchronization algorithm. Apart from introducing the problems, this work surprised by showing that the tasks *can* actually be solved, despite the severe fault model. It also shows how the "synchronous version" of pulse synchronization can serve as a testing ground for algorithmic ideas, without the messy details of drifting clocks and uncertain communication delays, before adapting them into solutions to pulse synchronization. This was also a main motivation of the line of work [DHJ$^+$16, LRS15, LRS17] culminating in the recursive construction presented in this lecture [LR16]: at the time it was unknown if more efficient (in particular sub-linear time) self-stabilizing solutions to pulse synchronization could be achieved, so we decided to study the synchronous counting problem as the "closest of kin" in the synchronous model.

Bibliography

[DFL+15] Danny Dolev, Matthias Függer, Christoph Lenzen, Ulrich Schmid, and Andreas Steininger. Fault-tolerant Distributed Systems in Hardware. *Bulletin of the EATCS*, 116, 2015.

[DHJ+16] Danny Dolev, Keijo Heljanko, Matti Järvisalo, Janne H. Korhonen, Christoph Lenzen, Joel Rybicki, Jukka Suomela, and Siert Wieringa. Synchronous Counting and Computational Algorithm Design. *Journal of Computer and System Sciences*, 82(2):310–332, 2016.

[DW04] S. Dolev and J. L. Welch. Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults. *Journal of the ACM*, 51(5):780–799, 2004.

[HDD06] Ezra Hoch, Danny Dolev, and Ariel Daliot. Self-Stabilizing Byzantine Digital Clock Synchronization. In *Proc. 8th Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 350–362, 2006.

[LR16] Christoph Lenzen and Joel Rybicki. Near-Optimal Self-stabilising Counting and Firing Squads. In Borzoo Bonakdarpour and Franck Petit, editors, *Proc. Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 263–280, 2016.

[LRS15] Christoph Lenzen, Joel Rybicki, and Jukka Suomela. Towards Optimal Synchronous Counting. In *Proc. 34th Symposium on Principles of Distributed Computing (PODC)*, pages 441–450, 2015.

[LRS17] C. Lenzen, J. Rybicki, and J. Suomela. Efficient Counting with Optimal Resilience. *SIAM Journal on Computing*, 46(4):1473–1500, 2017.

Lecture 12

Pulse Synchronization

In this lecture, we finally address the task of self-stabilizing pulse synchronization. A trace is good from time t if starting from then (i.e., when starting to count pulses from time t), the requirements of pulse synchronization are met.

We will head for an algorithm with a very good trade-off between stabilization time and communication complexity — here measured by the amount of bits a correct node broadcasts within $\mathcal{O}(d)$ time — right away. This will be achieved by reducing the task to consensus, very similar to the previous lecture. However, the fact that timing is now imprecise (due to uncertain message delays and drifting clocks), the details become rather complex. We will focus on the key ideas in this lecture, deliberately avoiding to give detailed proofs. Once we strip away these obfuscating issues, very little conceptual differences remain between the solution from the previous lecture and the algorithm presented today.

12.1 Outline of the Construction

We will sketch the overall construction, relying on the high similarity to the recursive construction of synchronous counting algorithms from the previous lecture for intuition. This will lead to identifying the main challenge in the approach, which we will focus on afterwards. Let us first state the final result of the machinery.

Theorem 12.1. *For $f \in \mathbb{N}_0$, denote by $C(f)$ (synchronous deterministic) consensus algorithms tolerating f faults on any number $n \geq 3f+1$ of nodes and by $R(f)$ and $M(f)$ their round complexities and message sizes, respectively. If $1 < \vartheta \leq 1.004$, there exists $T_0 \in \Theta(R(f))$ and $\varphi \in 1 + \mathcal{O}(\vartheta - 1)$ such that for any $T \geq T_0$ there is a pulse synchronization algorithm P satisfying that*

- *it stabilizes in $S(P) \in \mathcal{O}(d(1 + \sum_{k=0}^{\lceil \log f \rceil} R(2^k)))$ time,*
- *correct nodes broadcast $M(P) \in \mathcal{O}(1 + \sum_{k=0}^{\lceil \log f \rceil} M(2^k))$ bits per d time,*
- *it has skew $2d$,*
- *it has minimum period $P_{\min} \geq T$, and*
- *it has maximum period $P_{\max} \leq \varphi T$.*

Corollary 12.2. *If $1 < \vartheta \leq 1.004$, there exists $T_0 \in \Theta(f)$ and $\varphi \in 1 + \mathcal{O}(\vartheta - 1)$ such that for any $T \geq T_0$ there is a pulse synchronization algorithm P satisfying that*

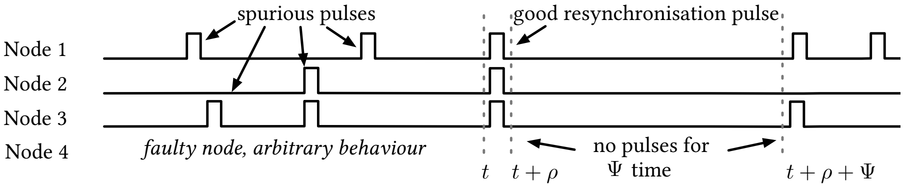
- *it stabilizes in $\mathcal{O}(df)$ time,*
- *correct nodes broadcast $\mathcal{O}(\log f)$ bits per d time,*
- *it has skew $2d$,*
- *it has minimum period $P_{\min} \geq T$, and*
- *it has maximum period $P_{\max} \leq \varphi T$.*

Proof. We plug the Phase King algorithm into Theorem 12.1. □

To make the recursion underlying this theorem work, again we need two main steps: The first is to construct pulse synchronization algorithms from *resynchronization algorithms*, which are "weak" pulse synchronization algorithms that produce a "proper" pulse only once in a while; the second is to construct resynchronization algorithms from two pulse synchronization algorithms on disjoint subsets of the nodes.

Definition 12.3 (Resynchronization Algorithm)**.** *B is an f-resilient* resynchronization algorithm *with skew ρ and separation window Ψ that stabilizes in time $S(B)$, if the following holds: there exists a time $t \leq S(B)$ such that every correct node $v \in V_g$ locally generates a* resynchronization pulse *at time $r(v) \in [t, t+\rho)$ and no other resynchronization pulse before time $t + \rho + \Psi$. We call such a resynchronization pulse* good.

Here is an example on how this might look like:



Note that we do not impose any restrictions on what the nodes do outside the interval $[t, t + \rho + \Psi)$. In particular, in constrast to the synchronous counting construction, we do not require that correct nodes agree on whether there are pulses or not outside this interval. Instead, this part of the construction will be subsumed by the first step, in which we construct pulse synchronization algorithms from resynchronization algorithms.

Using the same ideas as in the previous lecture, one can construct resynchronization algorithms from two smaller pulse synchronization instances as follows:

- Both instances may trigger resynchronization pulses via generating pulses.
- The instances (are supposed to) run at different frequencies. Hence, regardless of their initial phase relation, after a few pulses a correct instance (i.e., one with sufficiently many correct nodes) is guaranteed to produce a good pulse, provided that the other one adheres to its frequency bound.

- All correct nodes will “echo” seeing a pulse from either instance and only accept it if (i) $n-f$ nodes echoed the pulse, (ii) it adheres to the frequency bounds according to the node’s local clock, and (iii) the node didn’t recently observe fewer than $n-f$ and more than f nodes echo a pulse of the instance.

- If (i), (ii), or (iii) are violated, a node will (locally) suppress any pulses by the respective instance for sufficiently long to guarantee that the other (correct) instance succeeds in generating a good pulse.

As in the previous lecture, this forces a faulty instance to stick to the required frequency bound or be ignored entirely. It does not guarantee that all pulses are produced consistently (as we don’t run consensus), but this is not required from a resynchronization algorithm.

Once we also have a way of constructing pulse synchronization algorithms from resynchronization algorithms, which we will discuss in more depth, the recursive construction is performed exactly as for synchronous counting, cf. Figure 12.1. For $f=0$, pulse synchronization is trivial; all nodes simply trigger pulses when a designated leader tells them to. To construct an algorithm for $f \in [2^i, 2^{i+1}-1]$, $i \in \mathbb{N}$, faults, we select $f_0, f_1 < 2^i$ so that $f_0 + f_1 + 1 = f$ and (the already inductively constructed) pulse synchronization algorithms with $n_0 = 3f_0 + 1$ and $n_1 = 3f_1 + 1$ nodes, implying that $n_0 + n_1 < 3f + 1 \leq n$. From these we derive a resynchronization algorithm on all n nodes tolerating f faults, which in turn we use to obtain the desired pulse synchronization algorithm. Working out the details, one arrives at the result stated in Theorem 12.1.

Remarks:

- In Theorem 12.1, φ is a bit larger than ϑ, as the construction is lossy with respect to the quality of the hardware clocks. However, up to constant factors, the quality of the clocks is preserved: $\varphi - 1 \in \mathcal{O}(\vartheta - 1)$.

- The described construction of resynchronization algorithms is fraught with a frustating amount of bookkeeping due to the slightly different perception of time of the correct nodes. While the approach works just as described if $\vartheta \leq 1.004$ — a somewhat arbitrary bound that could be improved to a certain extent — formalizing the construction and proving it correct is very laborious.

- Accordingly, we will not do this in this lecture, but rather focus on the other main step of the recursive construction, in which we get to see some new algorithmic ideas.

12.2 Stabilization after Resynchronization Pulse

Before getting to business, let’s have a look at the general setting and a few notational simplifications. First, assume that at time 0 each node locally triggers a good resynchronization pulse, where Ψ “is large enough” for the stabilization process to finish before any other resynchronization pulse, good or bad, is triggered at a correct node (the minor time difference of up to δ between the resynchronization pulses can easily be accounted for, so we neglect it here). We need to guarantee that within Ψ time the algorithm stabilizes and cannot be

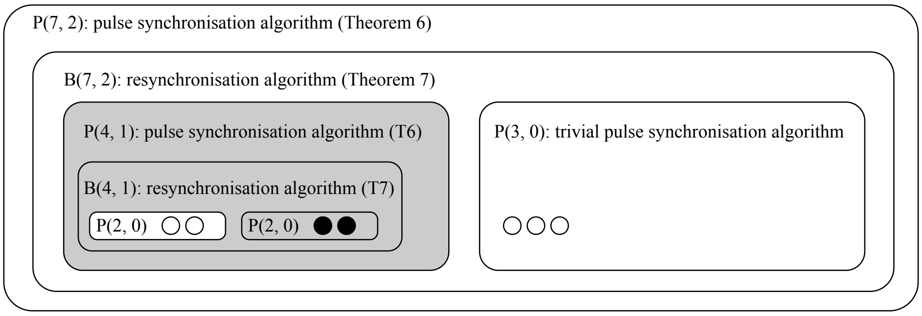

P(7, 2): pulse synchronisation algorithm (Theorem 6)
B(7, 2): resynchronisation algorithm (Theorem 7)
P(4, 1): pulse synchronisation algorithm (T6)
B(4, 1): resynchronisation algorithm (T7)
P(2, 0)
P(2, 0)
P(3, 0): trivial pulse synchronisation algorithm

Figure 12.1: Recursively building a 2-resilient pulse synchronization algorithm $P(7,2)$ over 7 nodes. The construction utilises low resilience pulse synchronization algorithms to build high resilience resynchronization algorithms, which can then be used to obtain highly resilient pulse synchronization algorithms. Here, the base case consists of trivial 0-resilient pulse synchronization algorithms $P(2,0)$ and $P(3,0)$ over 2 and 3 nodes, respectively. Two copies of $P(2,0)$ are used to build a 1-resilient resynchronization algorithm $B(4,1)$ over 4 nodes using. The resynchronization algorithm $B(4,1)$ is used to obtain a pulse synchronization algorithm $P(4,1)$. Now, the 1-resilient pulse synchronization algorithm $P(4,1)$ over 4 nodes is used together with the trivial 0-resilient algorithm $P(3,0)$ to obtain a 2-resilient resynchronization algorithm $B(7,2)$ for 7 nodes and the resulting pulse synchronization algorithm $P(7,2)$. White nodes represent correct nodes and black nodes represent faulty nodes. The gray blocks contain too many faulty nodes for the respective algorithms to correctly operate, and hence, they may have arbitrary output.

"confused" by any inconsistent resynchronization pulses. Accordingly, we will make sure that resynchronization pulses can affect the behavior of nodes only when the algorithm has not already stabilized.

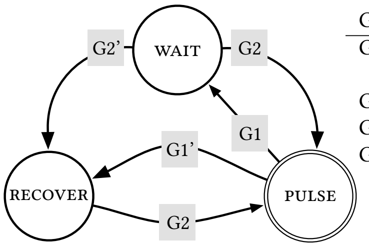
Our approach to generating pulses will be to execute consensus for each pulse. The challenge is to stabilize this procedure.

Simulating Consensus

We want to run synchronous consensus, but we're not operating in the synchronous model. Hence, we need to simulate synchronous execution. To this end, we may use any (non-stabilizing) pulse synchronization algorithm, where we locally count the pulses to keep track of the round number. This works splendidly, provided that each run is initialized correctly: using the Srikanth-Toueg algorithm (cf. Task 3 of exercise sheet 10), if all correct nodes start execution the pulse synchronization algorithm within a time window of $\mathcal{O}(Rd)$ time, simulation of an R-round consensus algorithm can be completed within $\mathcal{O}(Rd)$ time; the outputs will even be generated within $\mathcal{O}(d)$ time, as the skew of the algorithm is $2d$. We will never need more than one instance to run, so this will be efficient in terms of communication.

However, as nodes may initially be in arbitrary states, the simulation may get "messed up," at least until we can clear the associated variables and (re-)initialize them properly. This is the first challenge we need to overcome. In addition, we may also run into the familiar issue that not all correct nodes may *know* that they should simulate an instance. In this case, the pulse synchronization algorithm may not even function correctly. All of these problems will essentially be solved by employing silent consensus. Either all correct nodes participate, which causes them to reinitialize all state variables of the simulated consensus routine and ensures that the pulse synchronization algorithm works correctly, or no correct node will send messages for the consensus routine — meaning that it will never output 1 (the only result that matters), even if the simulation is completely off in terms of timing and attribution of messages to rounds due to the Srikanth-Toueg algorithm breaking. Summarizing, the simulation has the following properties

- Each node stores the state of at most one consensus instance. It aborts any local simulation if its local clock shows that it has been running for longer than the maximum possible time of $T_{\max} \in \mathcal{O}(Rd)$.
- If all correct nodes initialize an instance within $\tau \in \mathcal{O}(Rd)$ time (for a suitable relation between τ and $T_{\max}$) and none of them re-initialize for another instance, all correct nodes will terminate within $T_{\max}$ time and produce an output satisfying validity and agreement.
- If during $(t-d,t]$ no correct node is simulating an instance (i.e., by time t there are also no more respective messages in transit), no correct node will output a 1 as result of a simulation during $(t-d,t_1+T_{\min}]$, where t_1 is the infimal time larger than $t-d$ when a correct node initializes an instance with input 1 and $T_{\min} \in \Theta(Rd)$ is the minimum time to complete simulation of a consensus instance (note that we can enforce such a minimum time even for "incorrect" execution, by having nodes check the timing locally).



Guard	Condition
G1	$\langle T_1 \rangle$ expires and received $\geq n - f$ PULSE messages within time T_1 before T_1 expired
G1'	$\langle T_1 \rangle$ expires and $\neg$G1
G2	auxiliary machine signals OUTPUT 1
G2'	$\langle T_{\text{wait}} \rangle$ expires or auxiliary machine signals OUTPUT 0

Figure 12.2: The main state machine. When a node transitions to state PULSE (double circle) it will generate a local pulse event and send a PULSE message to all nodes. When the node transitions to state WAIT it broadcasts a WAIT message to all nodes. Guard G1 employs a sliding window memory buffer, which stores any PULSE messages that have arrived within time T_1 (as measured by the local clock). When a correct node transitions to PULSE it resets a local T_1 timeout. Once this expires, either Guard G1 or Guard G1' become satisfied. Similarly, the timer T_{wait} is reset when node transitions to WAIT. Once it expires, Guard G2' is satisfied and node transitions from WAIT to RECOVER. The node can transition to PULSE state when Guard G2 is satisfied, which requires an OUTPUT 1 signal from the auxiliary state machine given in Figure 12.3.

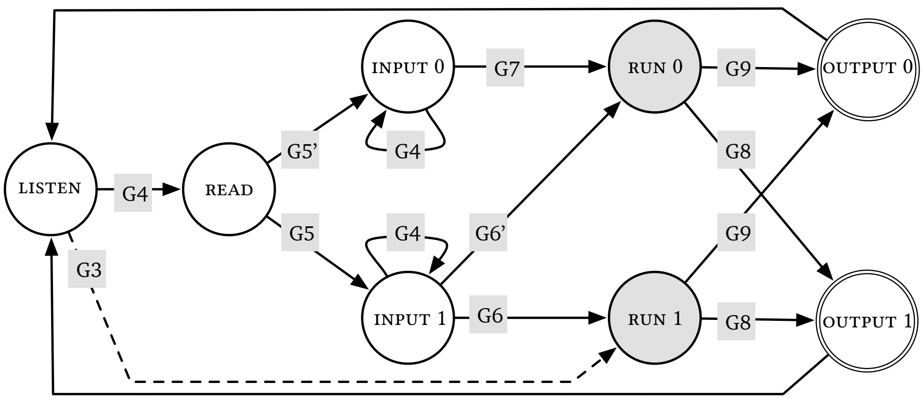
Remarks:

- A formal proof would require to work out the constants and how they relate to each other. However, as we know that ϑ is "sufficiently close" to 1, tweaking the period of the Srikanth-Toueg algorithm the right way, we can assume that $T_{\max} \approx T_{\min} + \tau$.

State Machines

Our overall strategy is simple. Once stabilized, the algorithm generates pulses by repeatedly executing consensus instances, where each correct node will use input 1, and an output of 1 triggers a pulse. To this end, each node runs a copy of the *main state machine* shown in fig. 12.2. All correct nodes will see each other generating a pulse within $T_1 \in \Theta(d)$ local time, transition to WAIT, and this will ultimately result in the next consensus instance being initialized by the *auxilliary state machine* (shown in fig. 12.3).

To achieve stabilization, we seek to enforce one of two events: either (i) a consensus instance is simulated correctly, outputs 1 (by agreement at all nodes), and thus generates a synchronized pulse kicking the system back into the intended mode of operation, or (ii) all nodes end up in state RECOVER of the state machine. A node being in state RECOVER means that it has proof that the algorithm has not stabilized (yet) and may thus take actions that are caused by a resynchronization pulse, as this does not jeopardize stable operation in case of spurious resynchronization pulses. Therefore, if we ensure that within $\mathcal{O}(dR)$ time after a good resynchronization pulse either (i) occurs or (ii) happens and no consensus instance is running anymore (or about to be started), we can "restart" the system by letting each correct node in state RECOVER start



Guard	Condition	Guard	Condition
G3	$\langle T_{\text{active}} \rangle$ expires while in RECOVER	G6	$\langle T_2 \rangle$ expires while not in RECOVER
G4	$\geq f + 1$ WAIT messages within time T_{listen}	G6’	$\langle T_2 \rangle$ expires while in RECOVER
G5	$\geq n - f$ WAIT messages within time T_{listen}	G7	$\langle T_2 \rangle$ expires
G5’	$\langle T_{\text{listen}} \rangle$ expires	G8	**A** outputs ‘1’
		G9	**A** outputs ‘0’or $\langle T_{\text{consensus}} \rangle$ expires or G4 is satisfied

Figure 12.3: The auxiliary state machine. The auxiliary state machine is responsible for initializing and simulating the consensus routine. The gray boxes denote states which represent the simulation of the consensus routine C. If the node transitions to RUN 0, it uses input 0 for the consensus routine. If the node transitions to RUN 1, it uses input 1. When the consensus simulation declares an output, the node transitions to either OUTPUT 0 or OUTPUT 1 (sending the respective output signal to the main state machine) and immediately to state LISTEN. The timeouts T_{listen}, T_2, and $T_{\text{consensus}}$ are reset when a node transitions to the respective states that use a guard referring to them. The timeout T_{active} in Guard G3 (dashed line) is reset by the resynchronisation signal from the underlying resynchronisation algorithm. Both INPUT 0 and INPUT 1 have a self-loop that is activated if Guard G4 is satisfied. This means that if Guard G4 is satisfied while in these states, the timer T_2 is reset.

a consensus instance with input 1, simply when a sufficiently large timeout of $\Theta(dR)$ expires.

Either way, a pulse with small skew will be generated, from which on the system will run as intended.

Lemma 12.4. *Suppose that all correct nodes transition to* PULSE *during* $[t, t+2d]$ *and timeouts are suitably chosen. Then the execution stabilized by time* t*, where a skew of* $2d$ *and period bounds of* $P_{\min} \geq (T_1 + T_2)/\vartheta + T_{\min}$ *and* $P_{\max} \leq T_1 + T_2 + T_{\max} + 3d$ *are guaranteed.*

Proof. Exercise. □

The challenge is to ensure that always one of the above two cases applies. This is mostly ensured by the design of the auxilliary state machine, which however takes into account the transitions to WAIT in the main state machine—which, in turn, does the consistency checks that (i) $n-f$ nodes should transition to PULSE within $T_1 \in \Theta(d)$ local time to go to WAIT and (ii) within $T_{\text{wait}} \in \Theta(dR)$ local time nodes expect to generate a pulse again. If either is not satisfied, the node transitions to RECOVER, which it leaves only when it generates a pulse again. A node in RECOVER knows that something is wrong and, accordingly, will use input 0 for consensus instances. The auxilliary state machine uses some additional thresholds based on transitions to WAIT.

Sketch of Proof of Stabilization

All these rules are designed to support the following line of reasoning:

1. Once the stabilization process is "started" by a resynchronization pulse, within $\mathcal{O}(dR)$ time no correct node will be executing consensus at some point (and no respective messages will be in transit).

2. From then on, any consensus instance outputting 1 must have been caused by a correct node transitioning to RUN 1, as otherwise the fact that the consensus routine is silent ensures that only output 0 can be generated (regardless of participation).

3. If any correct nodes transitions to RUN 1 before the (large) timeout T_{active} expires, all correct nodes will be "pulled" along into one of the input states (with suitable timing) to participate in the consensus instance, so it will be simulated correctly. Thus, if any node outputs 1, (i) applies.

4. On the other hand, if no correct node outputs a 1 for $\Theta(dR)$ time, they end up in the states RECOVER in the main state machine and LISTEN in the auxilliary state machine, i.e., case (ii) applies.

We now sketch proofs of these statements. Naturally, all of this hinges on the right choice of timeouts; to minimize distraction, in our proof sketch we will assume that they are suitably chosen. Moreover, T_{active}, which nodes reset upon locally triggering a resynchronization pulse, is "large enough," i.e., the dashed transition will not occur for long enough for us to either end up in case (i) (meaning that it will never happen) or case (ii) (meaning that the timeout expiring correctly initializes a consensus instance). To simplify matters further, assume that $\vartheta - 1$ is sufficiently small (read: a constant that is arbitrarily close to 1) and that all timeouts are in $\mathcal{O}(dR)$, which for such ϑ is feasible. Note that this implies that after all timeouts had the opportunity to expire, i.e., after $\mathcal{O}(rD)$ time, we know that timeouts and memory buffers of sliding windows are in states consistent with what actually happened, e.g., a node in state WAIT is there because it actually received $n-f$ PULSE messages within T_1 local time no more than T_{wait} local time ago.

We now work our way down the above list, starting by showing that, eventually, execution of consensus instances stops for at least d time at all correct nodes.

Lemma 12.5. *There is a time $t_0 \in \mathcal{O}(dR)$ such that no correct node is in states* RUN 0 *or* RUN 1 *during $[t_0, t_0+d]$.*

Proof Sketch. In order for any correct node to transition out of state LISTEN at some time t, there must be at least one correct node to transition to WAIT during $(t-\mathcal{O}(d), t]$. This, in turn, requires $n-2f$ correct nodes to transition to PULSE within $T_1+d \in \mathcal{O}(d)$ time. These nodes will have to get from state LISTEN in the auxilliary machine back to OUTPUT 1 again if they are to serve in supporting any correct node to transition to WAIT again. As $n > 3f$, the remaining correct nodes are not sufficiently many to reach the $n-f$ threshold for convincing a correct node to transition to WAIT, implying that for roughly (at least) T_2 time (see Guard G6, Guard G6', and Guard G7) no node can transition from READY to LISTEN.

Hence, no node leaving state LISTEN after time $t+\mathcal{O}(d)$ makes it to either of RUN 0 or RUN 1 before (roughly) time $t+2T_2$. On the other hand, any node that transitioned from LISTEN to READY by time $t+\mathcal{O}(d)$ will get back to LISTEN by time $t+\mathcal{O}(d)+T_{\text{listen}}+T_2+T_{\text{consensus}}$, where (as we will see later) $T_{\text{listen}} \in \mathcal{O}(d)$ and $T_{\text{consensus}} \approx T_{\max}$. Thus, up to minor order terms the claim follows if $T_2 > T_{\max}$, which can be arranged with $T_2 \in \mathcal{O}(dR)$.

Finally, observe that if no node transitions to READY for $\mathcal{O}(d)+T_{\text{listen}}+T_2+T_{\text{consensus}} \in \mathcal{O}(dR)$ time, then of course also all correct nodes end up in state LISTEN as well. □

Lemma 12.6. *Let t_0 be as in lemma 12.5. Suppose at time $t > t_0$, $v \in V_g$ transitions to* RUN 1*. Then each $w \in V_g$ transitions to* RUN 1 *or* RUN 0 *within a time window of size roughly $(1-1/\vartheta)T_2 + \mathcal{O}(d)$.*

Proof Sketch. We already observed that if any node transitions to WAIT, this means that there is a window of size $\mathcal{O}(d)$ during which this is possible, followed by a window of size at least T_2 during which this is not possible. Hence, in order for v to transition to RUN 1, it observes at least $n-2f > f$ correct nodes transition to WAIT within $\mathcal{O}(d)$ time (we make sure that T_2 is large enough to enforce this). This means that *all* correct nodes observe these transitions in a (slightly larger) time window. If we choose T_{listen} to be ϑ times this time window (i.e., still in $\mathcal{O}(d)$ as promised), this implies that (i) any correct node in state LISTEN, RUN 0, or RUN 1 transitions to READ and (ii) any node in states INPUT 0 or INPUT 1 resets its timeout T_2. As $T_{\text{listen}} \in \mathcal{O}(d)$, all of these nodes thus will, up to a time difference of $\mathcal{O}(d)$, switch to one of execution states after T_2 expires at them, i.e., within a time window of the required size. Here we use that the (properly initialized) consensus instance will not terminate and have nodes transition to PULSE (and thus potentially WAIT) again before its execution, i.e., the established timing relation between the nodes starting to execute consensus cannot be destroyed by another correct node switching to WAIT again. □

Corollary 12.7. *Let t_0 be as in lemma 12.5. If after time t_0 (but before any T_{active} timeout expires) any correct node transitions to* PULSE*, the system stabilizes.*

Proof Sketch. By lemma 12.5 and the fact that the utilized consenus routine is silent, after time t_0 no correct node can transition to PULSE without some correct node transitioning to RUN 1 first. By lemma 12.6, such an event will correctly initialize a consensus instance, which will thus be correctly simulated (note, again, that no transitions to WAIT happen before the instance terminates). If it outputs 1 (at all nodes), lemma 12.4 proves stabilization. If it outputs 0, we have a new time t'_0 such that no consensus instance is running and can repeat the argument inductively. □

Lemma 12.8. *Let t_0 be as in lemma 12.5. If by time $t_0 + \mathcal{O}(dR)$ the system has not stabilized, all correct nodes are in states* RECOVER *and* LISTEN*, with no* WAIT *messages in transit.*

Proof Sketch. If after time t_0 (but before T_{active} expires) any correct node outputs 1 for a consensus instance, then Corollary 12.7 shows stabilization. If this is not the case, no correct node transitions to PULSE. In this case, after $T_1 + T_{\text{wait}}$ time all correct nodes will be and stay in state RECOVER (without WAIT messages in transit), and after at most another $2T_{\text{listen}} + T_2 + T_{\text{consensus}} \in \mathcal{O}(dR)$ time all correct nodes will be in state LISTEN. As mentioned earlier, in all of this we assume that $T_{\text{active}} \in \mathcal{O}(dR)$ is large enough for this entire process to be complete before it expires at any correct node. □

Corollary 12.9. *The algorithm given by the state machines in fig. 12.2 and fig. 12.3 stabilizes within $\mathcal{O}(dR)$ time after a good resynchronization pulse, provided $\Psi \in \mathcal{O}(dR)$ is large enough.*

Proof Sketch. If the prerequisites of lemma 12.8 are not satisfied, the claim is immediate. Otherwise, when T_{active} expires at the correct nodes, they will transition from LISTENto RUN 1, as the lemma states that they are all in RECOVER and LISTEN. Correct initialization necessitates $\tau \geq (1 - 1/\vartheta)T_{\text{active}} \in \mathcal{O}((\vartheta - 1)dR)$, which is feasible for sufficiently small ϑ. Thus, for an appropriate choice of τ, the instance will be correctly simulated, and by validity it outputs 1. lemma 12.4 hence shows stabilization by time $T_{\text{active}} + T_{\max} \in \mathcal{O}(dR)$. □

Remarks:

- When actually proving this, one collects all the inequalities necessary for the various lemmas and then shows that there are assignments to the timeouts satisfying all of them concurrently.

- The framework can also be applied to randomized consensus routines. This way, it is, e.g., possible to get stabilization time of $\mathcal{O}(\log^2 f)$ (with high communication cost) or both stabilization time and broadcasted bits per d time both $\log^{\mathcal{O}(1)} f$ (with resilience $f < n/(3+\varepsilon)$ for arbitrarily small constant $\varepsilon > 0$).

- It is unknown whether any of these bounds are close to optimal. In contrast to synchronous counting, there is no reduction from consensus to self-stabilizing pulse synchronization known. For instance, it cannot be ruled out that a constant-time deterministic solution exists.

- We are still interested in showing how to combine this solution with Lynch-Welch along the lines of chapter 9. This will be done in an exercise.

Bibliographic Notes

We already mentioned that self-stabilizing pulse synchronization was first solved by Dolev and Welch [DW04], albeit with exponential stabilization time. Subsequently, this was improved to polynomial [**?**] and, eventually, linear [DH07]. The latter solution can be seen as the pulse synchronization equivalent of the counting algorithm derived from running R consensus instances concurrently — although here it is not one instance per round of the algorithm, but rather $\Theta(f)$ instances, the idea being that each node may initiate an instance, and this is going to succeed in stabilizing the algorithm, if the initiating node is correct.

This linear-time linear bandwidth (i.e., number of broadcasted bits per d time) was first overcome by randomization [DFLS14]. The idea of resynchronization pulses already shows up in this algorithm, but they are not provided recursively. Rather, each node may trigger a pulse once every $\Theta(fd)$ time by a simple broadcast, and randomization together with bounding the influence of faulty nodes by threshold voting and memorization ensures that this succeeds with a very large probability within $\mathcal{O}(fd)$ time. This improved the number of bits nodes need to broadcast for stabilization in $\mathcal{O}(df)$ time to $\mathcal{O}(1)$. However, the construction cannot be used recursively as-is, since the algorithm exploits that the resynchronization pulses are distributed randomly to avoid "bad" timing relations. The construction presented in this lecture [**?**] overcomes this restriction by relying on consensus.

It is worth noting that most constructions, in particular those of with stabilization time $\mathcal{O}(df)$, end up directly using consensus or tools that are strong enough to solve consensus in constant expected time. However, it remains open whether this is actually necessary or there are algorithms that outperform any consensus-based solution.

Bibliography

[DFLS14] Dolev Dolev, Matthias Függer, Christoph Lenzen, and Ulrich Schmid. Fault-tolerant Algorithms for Tick-generation in Asynchronous Logic. *Journal of the ACM*, 61(5):30:1–30:74, 2014.

[DH07] Danny Dolev and Ezra N. Hoch. Byzantine Self-stabilizing Pulse in a Bounded-Delay Model. In *Proc. 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems (2007)*, pages 234–252, 2007.

[DW04] S. Dolev and J. L. Welch. Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults. *Journal of the ACM*, 51(5):780–799, 2004.

[LR17] Christoph Lenzen and Joel Rybicki. Self-stabilising Byzantine Clock Synchronisation is Almost as Easy as Consensus. In *Proc. 31th Symposium on Distributed Computing (DISC)*, pages 32:1–32:15, 2017.

Lecture 13

Clock Distribution

For most of this lecture series, we have focused on fully connected topologies. As discussed, this has some justification, as high resilience to (permanent) faults requires high connectivity. However, in many cases it may simply not be practical to have a fully connected system — e.g., having $n(n-1)/2$ links on a chip means that we quickly run out of (physical) space for all these wires!

Instead, we can use the fault-tolerance techniques provided so far to build a reliable clock source, and strive for a fault-tolerant distribution scheme. Here, we need to assume that the (permanent) faults in the system are not distributed in a worst-case fashion, but reasonably "spread out." With this assumption, we can hope for smaller degrees, as we won't end up with a situation in which the neighborhood of a correct node is "taken over" by a majority of faulty ones, effectively cutting the node off from the rest of the network. In the following, we assume that each neighborhood contains at most f faults (for some constant or at least small f).

Definition 13.1 (Local Faults). *We say that there are at most f* local faults *in a given graph (V,E) with correct nodes $V_g \subseteq V$, if each $v \in V_g$ has at most f neighbors in $V \setminus V_g$. If the graph is directed, we require that at most f in-neighbors are faulty.*

This lecture is mostly about discussing this fairly open problem. We give no definite solutions, and we will not formalize the presented ones carefully.

13.1 First Attempt: Clock Trees

We could build a fault-tolerant version of a clock distribution tree. Using the techniques we discussed in previous lectures, we can set up a highly robust clock source serving as root of the tree, and we replace each node in the tree by a *cluster* of $2f+1$ nodes, whose (logical) clocks will just mirror what happens at the parent "node" in the tree. Concretely, this means to locally trigger and forward a clock flank — which we will refer to as *pulse* — when receiving the $(f+1)^{th}$ signal from the parent cluster. We end up with a very simple structure and node degrees of $\mathcal{O}(f)$, which is the best we can hope for. Self-stabilization is essentially for free, as we have a simple master-slave relationship; once the parent cluster pulses in synchrony and with the right frequency, its easy for the child cluster to start following suit.

Lemma 13.2. *If each node is faulty with uniform and independent probability of $p \in o(1/(fn^{f+1}))$ and (in-)degrees are at most $2f+1$, there are at most f local faults with probability $1-o(1)$.*

Proof. W.l.o.g., we assume that all nodes have in-degree $2f+1$, as the probability for having more than f faulty neighbors decreases with the degree (and all random choices are independent). For a single node, the probability that more than f of its neighbors are faulty is bounded by

$$\begin{aligned}\sum_{f'=f+1}^{2f+1}\binom{2f+1}{f'}p^{f'}(1-p)^{2f+1-f'} &\leq (f+1)\binom{2f+1}{f+1}p^{f+1}\\ &\leq ((2f+1)p)^{f+1} \in o\left(\frac{1}{n}\right).\end{aligned}$$

By the union bound, the overall probability of having more than f local faults is thus bounded by $\sum_{i=1}^{n} o(1/n) = o(1)$. □

Corollary 13.3. *Assume that each node is faulty with uniform and independent probability of $p \in o(1/(fn^{f+1}))$ and other system parameters (like delay and uncertainty) are identical to the clock tree from which the "fat tree" described above is derived. Then, with probability $1-o(1)$, each correct node produces clock pulses within the same worst-case time bounds as the corresponding node in the original tree.*

Proof. By Lemma 13.2, with probability $1-o(1)$ there are at most f local faults. Thus, any (non-root) node will disregard faulty nodes' signals unless they lie within the interval spanned by correct nodes' signals. As delays, etc. match those of the original tree, the resulting worst-case bounds are identical. □

Is the resulting solution good? It depends on the the system, but we can argue that it doesn't scale well. We know from the first lecture that the (worst-case) skew between tree nodes is proportional to their distance in terms of the considered graph. Sticking to (traditional) computer chips, a clock tree needs to provide the clock signal to a roughly quadratic area, where we can expect that at the very least physically close-by parts of the chip need the signals provided to them to be well-synchronized. At least for some nodes, a tree must fail to do so.

Lemma 13.4. *For $k \in \mathbb{N}$, consider a $k \times k$ grid in $\mathbb{R}^2$ in which adjacent grid nodes have distance 1. For any tree spanning the grid points, there are adjacent grid nodes that are in distance $\Omega(k)$ in the tree.*

Proof. Observe that if a tree node v has degree larger than 3, we can reduce its degree by inserting an additional node arbitrarily close to it and attaching 2 or more of the children of v to the new node instead. This changes tree distances by an arbitrarily small amount. Accordingly, we can w.l.o.g. assume that the tree is binary.

In any tree T of maximum degree 3 (of at least 4 nodes), there is some edge so that the two components resulting from removing this edge have size at least $(n-1)/3 \in \Omega(n)$. To see this, pick an arbitrary node edge, delete it, and look at the resulting components T_1 and T_2. If $|T_1|, |T_2| \geq (n-1)/3$, we're done.

Otherwise, assume w.l.o.g. that $|T_1| < (n-1)/3$, i.e., $|T_2| \geq n-(n-1)/3 = 2(n-1)/3+1$. Let w be the endpoint of the deleted edge that lies in T_2. Deleting w from T_2 results in (at most) two components of T_2, as w has degree 3 (and thus at most 2 in T_2). One of these components must have size at least $(|T_2|-1)/2 \geq (n-1)/3$. Consider the edge connecting w to this component and delete it from T, resulting in components T_1' and T_2'; let's say $w \in T_1'$. By the previous considerations, we have that $|T_1'| > |T_1|$, while $|T_2'| \geq (n-1)/3$. Now either $|T_1'| \geq (n-1)/3$ and we're done, or we can repeat the argument, resulting in an edge for which one component is even larger than T_1' and the other remains of size at least $(n-1)/3$. Thus, repeating this argument inductively, we must eventually reach an edge satisfying the claim.

From the above claim, we can infer that we can partition the nodes into two sets such that (i) each set contains $\Omega(k^2)$ nodes and (ii) each set induces a subtree. We call a node a *boundary node* if it has a neighbor in the other set. From (i) we can infer that the boundary must contain nodes in distance $\Omega(k)$ from each other, as any area of size $\Omega(k^2)$ must have a boundary of size $\Omega(k)$ (even when not considering nodes at the boundary of the entire grid). Fix two such nodes v and w in the same subtree. As they are in distance $\Omega(k)$, the path in the subtree connecting them is of that length. This in turn means that at least one of them is in distance $\Omega(k)$ of the root of its subtree. Finally, we conclude that this node is in distance $\Omega(k)$ from its neighbor(s) in the other set within the tree. □

Remarks:

- If uncertainties are proportional to the length of a link, we can immediately conclude that the worst-case skew between adjacent nodes is $\Omega(uk)$, cf. Theorem 1.6.

- One can rely on probabilistic guarantees instead. However, even if things behave "nicely," we still end up adding up variances in link delays, resulting in standard deviation $\Omega(u\sqrt{k})$ (if a unit length link has standard deviation u).

- There are quite a few tricks electrical engineers came up with in order to deal with the (few) long links of an H-tree (and relatives) in a better way, but ultimately physics gets in the way of scaling tree topologies arbitrarily.

- The above bound is tight up to constants, which is shown by an H-tree. For $k+1$ (the "width" of the grid) being a power of 2, an H-tree is constructed recursively as follows. Place the root in the center of the grid. Then connect it to two children by going $k/2$ to the right and left, respectively. Each of these children also has two children, which are in distance $k/4$ going up or down respectively. The four nodes in depth two of the tree are now exactly in the center of four disjoint subgrids of $k/2 \times k/2$ nodes, and the construction is applied recursively for $\log k$ steps. In the end, each grid point with both x- and y-index being odd (or even, depending on indexation) is occupied by a leaf.

- The construction from the lemma actually shows that there must be $\Omega(k)$ adjacent grid nodes that are in distance $\Omega(k)$ in the tree. More generally, one can show that $\Omega(2^i k)$ adjacent grid nodes are in distance $\Omega(k/2^i)$ in the tree.

- An H-tree matches this bound, too: Cutting the square in half horizontally and vertically, one gets four subsquares, each of which hosts a smaller H-tree. Where we cut the grid, we have $2k$ node pairs that end up being in distance (almost) $2k$ in the tree. All other node pairs are in the same of one of the four sub-H-trees, so they are by factor 2 closer to each other.

- All this is only useful when faults do not "cluster" within, well, clusters. The assumption that faults are completely independent is unrealistic, but it's crucial to design the system to make this an approximate reality. Of course, this also applies to the techniques from earlier chapters — a system-wide power failure will bring down any algorithm, regardless of how many node failures it can tolerate.

- Fault-tolerance and self-stabilization (under the assumption of at most f local faults) become easy due to the fact that we don't have any *cyclic* dependencies. In other words, we don't have to restrict ourselves to trees — DAGs are perfectly fine!

13.2 Second Attempt: Lynch-Welch on DAGs

The same strategy we used in Chapter 5 can be extended to directed acyclic graphs. Assuming f-local faults, we make sure that each node (except "source" nodes, which are supplied with a clock signal), have at least $3f+1$ in-neighbors. We then get the following property.

Corollary 13.5. *For $v, w \in V_g$, assume that both of them have the same set N of in-neighbors, where $|N| \geq 3f+1$ and $|V_g \cap N| \geq |N| - f$. Suppose each node $x \in N_g := V_g \cap N$ pulses once at time p_x and v, w interpret the respective messages as in Line 5 and that T and $\mathcal{S}$ are sufficiently large. Then v and w pulse at times p_v and p_w such that $|p_v - p_w| \in \mathcal{O}(\|\vec{p}\|/2 + (\vartheta - 1)T + u)$.*

- We can use this to adapt the earlier approach of fat trees to guarantee that local skews *with respect to the tree topology* and any given pulse remain bounded, regardless of the depth of the tree. However, this does not fix the problem of large skews between different branches of the tree.

- As stated, we can use this on more general DAGs. We could, e.g., take two clusters and let them have a "common" child. In this case, the $3f+1$ nodes of the child cluster may exhibit a larger skew, though, as the bound on the skew between the two parent clusters may be much larger than within each parent cluster.

- This leads to the following open problem: Is there a clever choice of a DAG that avoids the issues of trees, i.e., it can clock a two- (or three-)dimensional area with small local skews (with respect to physical distances)?

13.3 Third Attempt: Fault-tolerant GCS

Trying something else, we could seek to simulate the (non-fault-tolerant) GCS algorithm from chapter 2 on an arbitrary topology. In fact, node degrees of 3 are sufficient to cover any structure without "misrepresenting" physical distances too much.

The general idea is to have clusters of size $3f+1$, which each may have up to f faulty nodes, that we synchronize internally with the Lynch-Welch algorithm. However, these clocks will not be the hardware clocks of the GCS algorithm — they will be the logical clocks! Concretely:

- The output of the Lynch-Welch algorithm at a (correct) node can be seen as (up to the skew) accurate representation of the cluster's logical clock. The node will communicate this clock to all adjacent clusters.

- At the same time, it is the node's logical clock of the Lynch-Welch algorithm, which is set up to handle the increased clock drift of a logical clock of the GCS algorithm, i.e., it assumes that the underlying "hardware" clock has drift $\vartheta(1+\mu)$.

- The GCS algorithm also takes into account a larger "hardware" clock drift, as it needs to account for the clock corrections of the Lynch-Welch algorithm within clusters. By amortizing these changes over $\Theta(T)$ time, this implies a "hardware" clock drift of $\mathcal{O}(\vartheta(1+\mu)+u/T)$, and by choosing T large enough and recalling that $\mu \in \mathcal{O}(1)$, we can bound this by $\mathcal{O}(\vartheta)$. In other words, if ϑ is small enough, both algorithms are able to handle each other's clock manipulations.

- Now each node can read the own cluster clock (by looking at its own clock) and that of adjacent clusters (by looking at all nodes there and, e.g., taking the median value) with an error of $u+\mathcal{S}$ (plus possibly a term for refreshing this information only infrequently), where $\mathcal{S} \in \mathcal{O}((\vartheta-1)d+u)$ is the skew of the Lynch-Welch algorithm within each cluster.

- This yields a $\delta \in \mathcal{O}((\vartheta-1)d+u)$ for the GCS algorithm, i.e., the same asymptotic guarantee as for the original GCS algorithm.

- To control the global skew, one could, e.g., use a tree structure as discussed above, employing the strategy from Task 1 of the second exercise sheet.

Remarks:

- Take this description with a grain of salt; it's an idea for an approach that has not been proven correct yet.

- The failure condition of the system (more than f faults in a cluster) is slightly different from f-local faults, but the asymptotics in terms of the failure probability that can be sustained are the same.

- Even for $f=1$ and a maximum degree of 3 of the original graph, the minimum node degree is $3f+3(3f+1)=12f+3=15$, where all these links are bidirectional. One could reduce this to $3f+3(2f+1)=12$ by arguing that $2f+1$ values suffice to "read" a cluster clock (the median

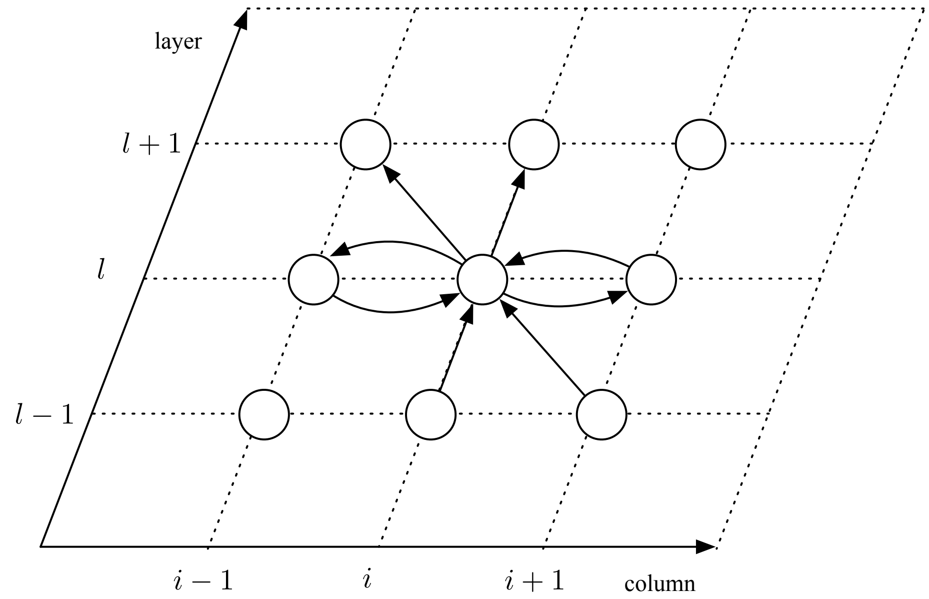


Figure 13.1: Structure of the HEX grid.

value will be in the range of correct nodes' values), but this comes at the cost of having less accurate readings.

- If all of this works out, efficiently adding self-stabilization is still an open problem. We have discussed how to do this for the Lynch-Welch algorithm, but we also need to make sure that clock values are communicated correctly between clusters. At least when considering a hardware implementation, communicating (and performing computations with) encoded clock values, as opposed to just sending clock "ticks," might be rather expensive.

- When all of this is done, despite the good asymptotics, we end up with awfully complicated nodes. Even if we can afford the energy and area for this, it means that our nodes are more likely to fail than simpler ones. The gains in reliability may be small or we may even end up with a *less* reliable system! We may need much simpler solutions!

13.4 Fourth Attempt: HEX

With this issue in mind, we looked for a very low-degree (for convenience also planar) topology that can tolerate one local fault, with an extremely simple algorithm.

The idea is that nodes are organized in layers, where the nodes in the initial layer are provided with a clock signal by some other means. A node locally triggers and forwards a pulse when it has received messages from two in-neighbors. Byzantine nodes cannot trigger spurious faults if we have 1-local faults. And they also can't prevent nodes from triggering their pulse, as an in-neighbor in the same layer will provide a pulse signal in case an in-neighbor in the previous layer is faulty (this is not completely trivial, but easy to show).

With reasonable effort, the approach can also be made self-stabilizing, by ensuring via some timeouts that the system will recover layer by layer once the clock source works correctly. There are also elaborate (laborious?) proofs showing that HEX has a worst-case skew of $d + \mathcal{O}(\min\{L, W\} \cdot u^2/d)$ between neighbors in the same layer, where L is the number of layers and W the "width" of the grid, respectively, granted that skews at initial layer are 0 (non-zero skews are accounted for by the bounds as well). Moreover, the same bound holds between neighbors from different layers when shifting the output clocks by an additive d per layer to account for delays. Despite all this, HEX suffers from a rather basic flaw.

Observation 13.6. *Even with a single crash fault, perfect input (i.e., skew* 0 *on the initial layer), and* $u = 0$*, correct neighbors in the same layer may exhibit a skew of* d*.*

Proof. The crashing node will cause its out-neighbors on the next layer to trigger a pulse d time later, yet all other nodes on this layer will pulse at the same time as they would without the fault. □

You might argue now that the bound we had already has an additive d in it, and you would be correct — from the point of view of a worst-case analysis. In contrast, things look very different when assuming that delays do not behave in a worst-case fashion.

Getting a better understanding of this is an art. First, one needs to determine how delays behave if they are not chosen adversarially (meaning that we just make sure that this does not matter). As a first stab at the task, one may be optimistic: we assume that each delay is chosen independently and uniformly at random from $(d - u, d)$. Second, the resulting distributions appear to be very hard to analyze mathematically. This is owed to the fact that they are the result of operations involving both taking the minimum and maximum, disqualifying straightforward application of concentration bounds and other basic probabilistic tools.

We ended up simply study the grid under the above assumptions by computer simulation. The result was that, if there are no faults (and skews at the initial layer are small), the grid performs astoundingly well, without ever needing any of the communication links between nodes in the same layer, see Figure 13.2. We observe skews of $\mathcal{O}(u)$, where $u \ll d$ implies that the worst-case bounds are not all that informative. In a way, the grid performed *too well* for the approach to fault-tolerance to make sense!

Remarks:

- The name "HEX" originates in the hexagonal structure of a node's neighborhood in what appears to be the most natural physical layout.

13.5 Fifth Attempt: TRIX

The above insights suggest an even simpler topology, in which each node has 3 in- and outneighbors each, and triggers a pulse when receiving the second pulse on its incoming links.

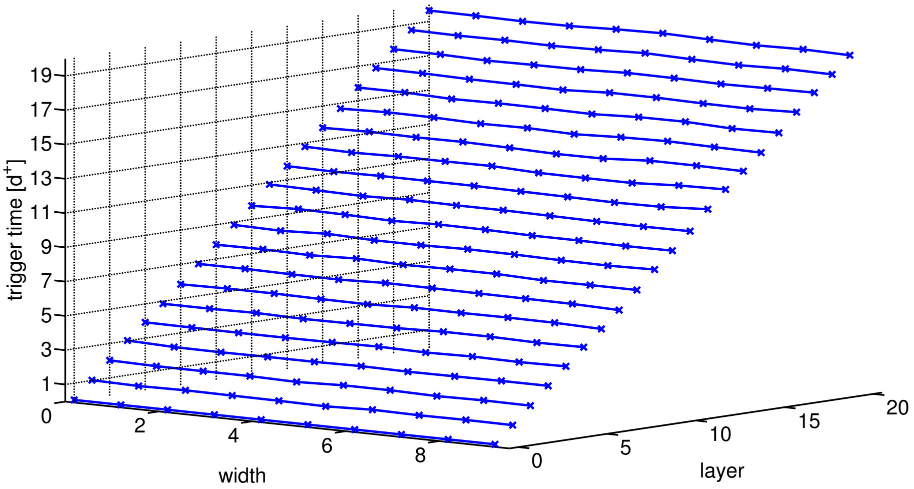


Figure 13.2: A pulse of a fault-free HEX grid with uniformly random delays ($d^+ = d$). It's easy to see that skews between neighbors remain far smaller than d, meaning that the links within a layer never contribute to triggering pulses.

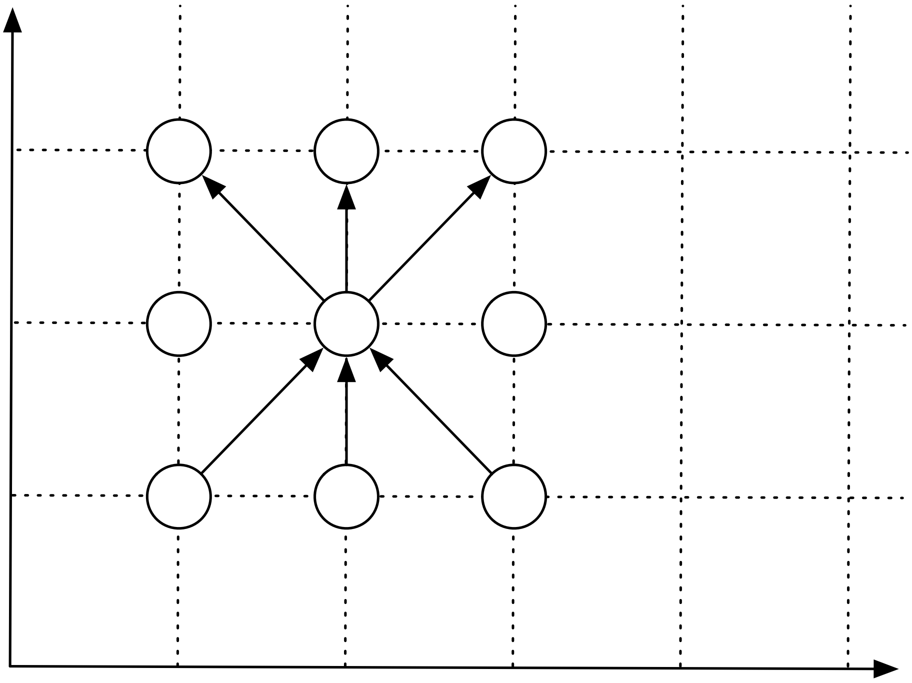


Figure 13.3: Structure of the TRIX grid.

Note that from the point of view of a worst-case analysis, this solution is rather bad: If all message delays are small on one "side" of the grid and large on the other, a skew of up to u per layer can be built up, which is by a factor of (up to reasonably small constants) d/u worse than HEX — except that HEX suffers from an additive d in its bound. Up to d/u layers, HEX thus doesn't really look better on this front, requiring at least 100 layers before these asymptotics could kick in.

But these are worst-case bounds, which the above considerations suggest to not matter as much as one might think. An important advantage of TRIX over HEX is that even when a node fails, this does not cause any of its out-neighbors to be triggered significantly later. As under the assumption of 1-local faults any node has two in-neighbors on the previous layer, it will still trigger its pulse between $d-u$ and d time after (at least) one of its correct in-neighbors.

In some ways, TRIX is much easier to analyze: self-stabilization is almost trivial, as there are no interactions between nodes in the same layer. The worst-case skew bounds are essentially obvious. The effect of faults on skews can be easily bounded by u for any given fault, and one can easily suspect that faults have no significant effect on far-away parts of the system (so long as they are 1-local). But when it comes to the behavior of skews under u.i.r. link delays, the situation is equally embarrassing as for HEX.

Here are some results from computer experiments, where for simplicity link delays are either 0 or 1 with (independent) probability of 1/2 each, and there is no skew in the first layer.

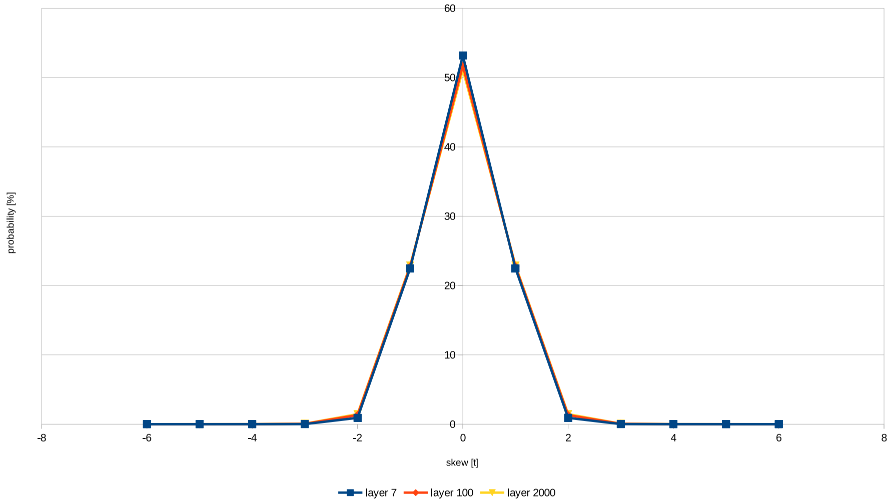


Figure 13.4: Distributions of skews between neighbors in layers 7, 100, and 2000 for TRIX with random 0-1-delays. This suggests a binomial (or related) distribution with extremly small standard deviation, where the number of layers has very limited influence.

Remarks:

- The name "TRIX" indicates that in- and outdegrees are three (and plays on "HEX").

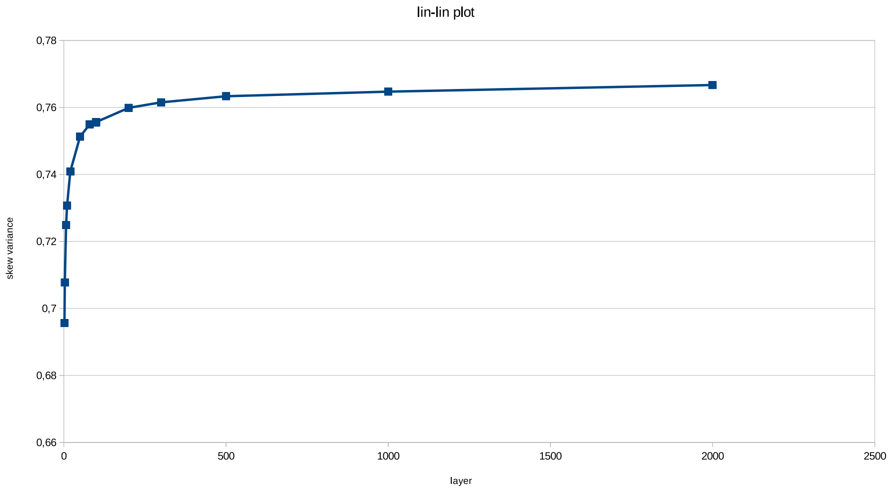


Figure 13.5: Plot of the variance of the above distribution as function of the number of layers. Does it grow very slowly, but is unbounded, or does it converge to a fixed value corresponding to a limit distribution?

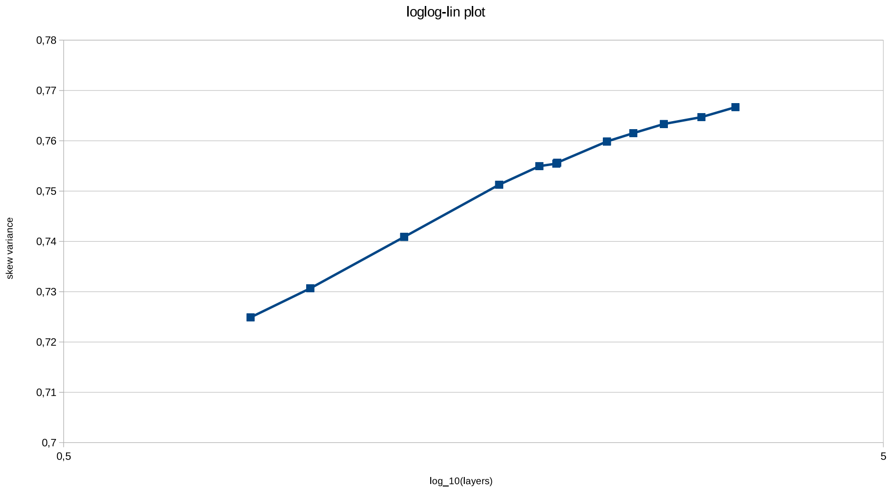


Figure 13.6: The same plot, but with a doubly logarithmic x-scale. One may suspect that the growth is bounded by $\log\log L$, where L is the number of layers. However, the considered range of values is rather small for this (especially given the very small change of the actual y-values), and the number of experiments may be insufficient for a reliable assessment.

- We do not understand why the TRIX grid appears to work so extraordinarily well with randomized link delays.

- Do the skews just grow very slowly, or converge to a limit distribution?

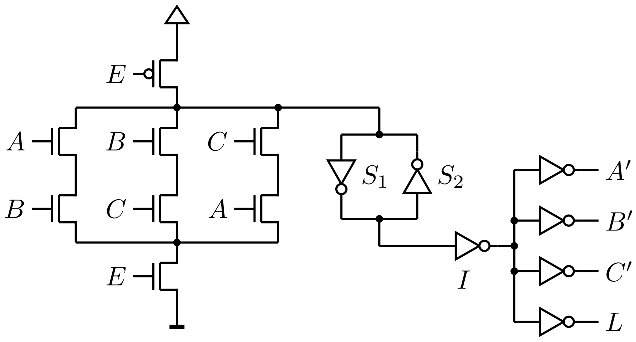


Figure 13.7: Circuit implementing the critical path in triggering a pulse. The "gates" connected to signals A, B, C, and E are transistors; you can think of them as switches that are "open" (i.e., conduct) if the input is high (a.k.a. 1) and "closed" (i.e., have almost infinite resistance) if the input is low (a.k.a. 0). E is an enable signal that is controlled in a non-time-critical manner by additional (local) circuitry. At the beginning of a cycle, enable is high, i.e., the connection to ground (in the bottom) is open and the connection to the supply voltage (top) is closed; the storage loop formed by the inverters S_1 and S_2 has low output, meaning that the outgoing (A', B', and C') and local (L) output signals are low. As soon as at least two out of the three incoming signals (A, B, and C) become high, this pulls the storage loop input to low (i.e., ground), and the outputs switch to high. The inverter I has a high threshold, meaning that S_1, S_2, and I essentially form a masking register. Thus, no matter what a faulty node provides as input, the output transition will be clean and occur within the interval spanned by the correct in-neighbors' signal transition times (plus delay). E is then controlled via L in a way that holds the output signals high for long enough for the outneighbors to react and for the incoming correct signals to go low again, and then is set to low in order to "reset" the circuit to the initial state, readying it for the next pulse.

- Is the grid even reducing larger skews effectively?
- For a practical realization, one has to further adapt the topology. Concretely, the clock source should not be a (wide) intial layer, but rather a few nodes. This suggests a layout with layers being nested circles, adding a constant number of nodes per layer. These and other concerns will affect a final design, requiring further studies.

TRIX and Metastability

TRIX nodes are extremely simple, but that doesn't mean that we can ignore metastability issues. Given that we assume Byzantine faults, we should better make sure that they don't break the abstract pulse triggering logic we've been thinking about. Synchronizers are not an option here, as (forcibly) aligning the incoming pulses to some local clock would significantly increase skews. There are unclocked circuit elements with similar function, but we can take a more direct approach by designing a suitable circuit element for the (time-)critical path on the transistor level.

Remarks:

- This is conceptually very similar to the Strategy shown in Figure 7.6, except for the addition of the loop capturing the transition. The storage loop ensures that a Byzantine node cannot, e.g., wait until the first correct signal transition arrived, then pull the output high (using its own input), and then pull it down again before the second correct transition arrives.

Bibliographic Notes

Clock trees have been the standard clocking method for decades, and they still are for many systems or at least subsystems. They have been heavily engineered, and in fact most clock "trees" are not real trees any more, for instance due to a grid-like structure connecting the leaves to reduce skews. It's difficult to find comprehensive and up-to-date literature, as most developments nowadays occur within companies, which are not eager to share their know-how. See [X09] for some introductory material.

To my knowledge, neither have variants of the Lynch-Welch algorithm on not fully connected graphs been studied, nor have attempts at making GCS algorithms fault-tolerant been made in the literature; fault-tolerant clock distribution methods — at least in terms of mathematical proven guarantees — appear to be essentially unchartered territory. For results on HEX see [DFL+16]; no publications discussing TRIX exist at the time of writing.

Bibliography

[DFL+16] Danny Dolev, Matthias Függer, Christoph Lenzen, Martin Perner, and Ulrich Schmid. HEX: Scaling Honeycombs is Easier than Scaling Clock Trees. *J. Comput. Syst. Sci.*, 82(5):929–956, 2016. Preprint available at https://people.mpi-inf.mpg.de/∼clenzen/pubs/DFLPS16scaling.pdf.

[X09] Thucydides Xanthopoulos. *Clocking in modern VLSI systems.* Springer Science & Business Media, 2009.

Appendix A

Notation and Preliminaries

This appendix sums up important notation, definitions, and key lemmas that are not the main focus of the lecture.

A.1 Numbers and Sets

In this lecture, zero is not a natural number: $0 \notin \mathbb{N}$; we just write $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$ whenever we need it. $\mathbb{Z}$ denotes the integers, $\mathbb{Q}$ the rational numbers, and $\mathbb{R}$ the real numbers. We use $\mathbb{R}^+ = \{x \in \mathbb{R} \mid x > 0\}$ and $\mathbb{R}_0^+ = \{x \in \mathbb{R} \mid x \geq 0\}$.

Rounding down $x \in \mathbb{R}$ is denoted by $\lfloor x \rfloor := \max\{z \in \mathbb{Z} \mid z \leq x\}$ and rounding up by $\lceil x \rceil := \min\{z \in \mathbb{Z} \mid z \geq x\}$.

For $n \in \mathbb{N}_0$, we define $[n] := \{0, \dots, n-1\}$, and for a set M and $k \in \mathbb{N}_0$, $\binom{M}{k} := \{N \subseteq M \mid |N| = k\}$ is the set of all subsets of M that contain exactly k elements.

A.2 Graphs

A finite set of *vertices*, also referred to as *nodes* V together with *edges* $E \subseteq \binom{V}{2}$ defines a *graph* $G = (V, E)$. Unless specified otherwise, G has $n = |V|$ vertices and $m = |E|$ edges and the graph is *simple*: Edges $e = \{v, w\} \subseteq V$ are undirected, there are no *loops*, and there are no *parallel edges*.

If $e = \{v, w\} \in E$, the vertices v and w are *adjacent*, and e is *incident* to v and w, furthermore, $e' \in E$ is *adjacent* to e if $e \cap e' \neq \emptyset$. The *neighborhood* of v is

$$N_v := \{w \in V \mid \{v, w\} \in E\},$$

i.e., the set of vertices adjacent to v. The *degree* of v is

$$\delta_v := |N_v|,$$

the size of v's neighborhood. We denote by

$$\Delta := \max_{v \in V} \delta_v$$

the maximum degree in G.

A v_1-v_d-*path* p is a set of edges $p = \{\{v_1, v_2\}, \{v_2, v_3\}, \dots, \{v_{d-1}, v_d\}\}$ such that $|\{e \in p \mid v \in e\}| \leq 2$ for all $v \in V$. p has $|p|$ *hops*, and we call p a *cycle* if it visits all of its nodes exactly twice. The *diameter* D of the graph is the minimum integer such that for any $v, w \in V$ there is a v-w-path of at most D hops (or $D = \infty$ if no such integer exists). We consider *connected* graphs only, i.e., graphs satisfying $D \neq \infty$.

A.2.1 Trees and Forests

A *forest* is a cycle-free graph, and a *tree* is a connected forest. Trees have $n-1$ edges and a unique path between any pair of vertices. The tree $T = (V, E)$ is *rooted* if it has a designated root node $r \in V$. A leaf is a node of degree 1. A rooted tree has *depth* d if the maximum length of a root-leaf path is d.

A.3 Asymptotic Notation

We require asymptotic notation to reason about the complexity of algorithms. This section is adapted from Chapter 3 of Cormen et al. [**?**]. Let $f, g \colon \mathbb{N}_0 \to \mathbb{R}$ be functions.

A.3.1 Definitions

$\mathcal{O}(g(n))$ is the set containing all functions f that are bounded from above by $cg(n)$ for some constant $c > 0$ and for all sufficiently large n, i.e. $f(n)$ is *asymptotically bounded from above* by $g(n)$.

$$\mathcal{O}(g(n)) := \{f(n) \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}_0 \colon \quad \forall n \geq n_0 \colon \quad 0 \leq f(n) \leq cg(n)\}$$

The counterpart of $\mathcal{O}(g(n))$ is $\Omega(g(n))$, the set of functions *asymptotically bounded from below* by $g(n)$, again up to a positive scalar and for sufficiently large n:

$$\Omega(g(n)) := \{f(n) \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}_0 \colon \quad \forall n \geq n_0 \colon \quad 0 \leq cg(n) \leq f(n)\}$$

If $f(n)$ is bounded from below by $c_1 g(n)$ and from above by $c_2 g(n)$ for positive scalars c_1 and c_2 and sufficiently large n, it belongs to the set $\Theta(g(n))$; in this case $g(n)$ is an *asymptotically tight* bound for $f(n)$. It is easy to check that $\Theta(g(n))$ is the intersection of $\mathcal{O}(g(n))$ and $\Omega(g(n))$.

$$\begin{aligned} \Theta(g(n)) := \{f(n) \mid \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N}_0 \colon \quad \forall n \geq n_0 \colon \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\} \\ f(n) \in \Theta(g(n)) \quad \Leftrightarrow \quad f \in (\mathcal{O}(g(n)) \cap \Omega(g(n))) \end{aligned}$$

For example, $n \in \mathcal{O}(n^2)$ but $n \notin \Omega(n^2)$ and thus $n \notin \Theta(n^2)$.[1] But $3n^2 - n + 5 \in \mathcal{O}(n^2)$, $3n^2 - n + 5 \in \Omega(n^2)$, and thus $3n^2 - n + 5 \in \Theta(n^2)$ for $c_1 = 1$, $c_2 = 3$, and $n_0 = 4$.

[1] We write $f(n) \in \mathcal{O}(g(n))$ unlike some authors who, by abuse of notation, write $f(n) = \mathcal{O}(g(n))$. $f(n) \in \mathcal{O}(g(n))$ emphasizes that we are dealing with *sets* of functions.

In order to express that an asymptotic bound is not tight, we require $o(g(n))$ and $\omega(g(n))$. $f(n) \in o(g(n))$ means that for any positive constant c, $f(n)$ is strictly smaller than $cg(n)$ for sufficiently large n.

$$o(g(n)) := \{f(n) \mid \forall c \in \mathbb{R}^+ \colon \exists n_0 \in \mathbb{N}_0 \colon \forall n \geq n_0 \colon \quad 0 \leq f(n) < cg(n)\}$$

As an example, consider $\frac{1}{n}$. For arbitrary $c \in \mathbb{R}^+$, $\frac{1}{n} < c$ we have that for all $n \geq \frac{1}{c} + 1$, so $\frac{1}{n} \in o(1)$. A similar concept exists for lower bounds that are not asymptotically tight; $f(n) \in \omega(g(n))$ if for any positive scalar c, $cg(n) < f(n)$ as soon as n is large enough.

$$\omega(g(n)) := \{f(n) \mid \forall c \in \mathbb{R}^+ \colon \exists n_0 \in \mathbb{N}_0 \colon \forall n \geq n_0 \colon \quad 0 \leq cg(n) < f(n)\}$$
$$f(n) \in \omega(g(n)) \quad \Leftrightarrow \quad g(n) \in o(f(n))$$

A.3.2 Properties

We list some useful properties of asymptotic notation, all taken from Chapter 3 of Cormen et al. [**?**]. The statements in this subsection hold for all $f, g, h \colon \mathbb{N}_0 \to \mathbb{R}$.

Transitivity

$$\begin{aligned}
f(n) \in \mathcal{O}(g(n)) \wedge g(n) \in \mathcal{O}(h(n)) &\quad\Rightarrow\quad f(n) \in \mathcal{O}(h(n)),\\
f(n) \in \Omega(g(n)) \wedge g(n) \in \Omega(h(n)) &\quad\Rightarrow\quad f(n) \in \Omega(h(n)),\\
f(n) \in \Theta(g(n)) \wedge g(n) \in \Theta(h(n)) &\quad\Rightarrow\quad f(n) \in \Theta(h(n)),\\
f(n) \in o(g(n)) \wedge g(n) \in o(h(n)) &\quad\Rightarrow\quad f(n) \in o(h(n)), \text{ and}\\
f(n) \in \omega(g(n)) \wedge g(n) \in \omega(h(n)) &\quad\Rightarrow\quad f(n) \in \omega(h(n)).
\end{aligned}$$

Reflexivity

$$\begin{aligned}
&f(n) \in \mathcal{O}(f(n)),\\
&f(n) \in \Omega(f(n)), \text{ and}\\
&f(n) \in \Theta(f(n)).
\end{aligned}$$

Symmetry

$$f(n) \in \Theta(g(n)) \quad \Leftrightarrow \quad g(n) \in \Theta(f(n)).$$

Transpose Symmetry

$$\begin{aligned}
f(n) \in \mathcal{O}(g(n)) &\quad\Leftrightarrow\quad g(n) \in \Omega(f(n)), \text{ and}\\
f(n) \in o(g(n)) &\quad\Leftrightarrow\quad g(n) \in \omega(f(n)).
\end{aligned}$$

A.4 Bounding the Growth of a Maximum of Differentiable Functions

Lemma A.1. *For $k \in \mathbb{N}$, let $\mathcal{F} = \{f_i \mid i \in [k]\}$, where each $f_i \colon [t_0, t_1] \to \mathbb{R}$ is differentiable, and $[t_0, t_1] \subset \mathbb{R}$. Define $F \colon [t_0, t_1] \to \mathbb{R}$ by $F(t) \coloneqq \max_{i \in [k]} \{f_i(t)\}$. Suppose $\mathcal{F}$ has the property that for every i and t, if $f_i(t) = F(t)$, then $\frac{d}{dt} f_i(t) \leq r$. Then for all $t \in [t_0, t_1]$, we have $F(t) \leq F(t_0) + r(t - t_0)$.*

Proof. We prove the stronger claim that for all a, b satisfying $t_0 \leq a < b \leq t_1$, we have

$$\frac{F(b) - F(a)}{b - a} \leq r. \tag{A.1}$$

To this end, suppose to the contrary that there exist $a_0 < b_0$ satisfying $(F(b_0) - F(a_0))/(b_0 - a_0) \geq r + \varepsilon$ for some $\varepsilon > 0$. We define a sequence of nested intervals $[a_0, b_0] \supset [a_1, b_1] \supset \cdots$ as follows. Given $[a_j, b_j]$, let $c_j = (b_j + a_j)/2$ be the midpoint of a_j and b_j. Observe that

$$\frac{F(b_j) - F(a_j)}{b_j - a_j} = \frac{1}{2}\frac{F(b_j) - F(c_j)}{b_j - c_j} + \frac{1}{2}\frac{F(c_j) - F(a_j)}{c_j - a_j} \geq r + \varepsilon,$$

so that

$$\frac{F(b_j) - F(c_j)}{b_j - c_j} \geq r + \varepsilon \quad \text{or} \quad \frac{F(c_j) - F(a_j)}{c_j - a_j} \geq r + \varepsilon.$$

If the first inequality holds, define $a_{j+1} = c_j$, $b_{j+1} = b_j$, and otherwise define $a_{j+1} = a_j$, $b_j = c_j$. From the construction of the sequence, it is clear that for all j we have

$$\frac{F(b_j) - F(a_j)}{b_j - a_j} \geq r + \varepsilon. \tag{A.2}$$

Observe that the sequences $\{a_j\}_{j=0}^{\infty}$ and $\{b_j\}_{j=0}^{\infty}$ ar both bounded and monotonic, hence convergent. Further, since $b_j - a_j = \frac{1}{2^j}(b_0 - a_0)$, the two sequences share the same limit.

Define

$$c \coloneqq \lim_{j \to \infty} a_j = \lim_{j \to \infty} b_j,$$

and let $f \in \mathcal{F}$ be a function satisfying $f(c) = F(c)$. By the hypothesis of the lemma, we have $f'(c) \leq r$, so that

$$\lim_{h \to 0} \frac{f(c + h) - f(h)}{h} \leq r.$$

Therefore, there exists some $h > 0$ such that for all $t \in [c - h, c + h]$, $t \neq c$, we have

$$\frac{f(t) - f(c)}{t - c} \leq r + \frac{1}{2}\varepsilon.$$

Further, from the definition of c, there exists $N \in \mathbb{N}$ such that for all $j \geq N$, we have $a_j, b_j \in [c - h, c + h]$. In particular this implies that for all sufficiently large j, we have

$$\frac{f(c) - f(a_j)}{c - a_j} \leq r + \frac{1}{2}\varepsilon, \tag{A.3}$$

$$\frac{f(b_j) - f(c)}{b_j - c} \leq r + \frac{1}{2}\varepsilon. \tag{A.4}$$

Since $f(a_j) \leq F(a_j)$ and $f(c) = F(c)$, (A.3) implies that for all $j \geq N$,

$$\frac{F(c) - F(a_j)}{c - a_j} \leq r + \frac{1}{2}\varepsilon.$$

However, this expression combined with with (A.2) implies that for all $j \geq N$

$$\frac{F(b_j) - F(c)}{b_j - c} \geq r + \varepsilon. \tag{A.5}$$

Since $F(c) = f(c)$, the previous expression together with (A.4) implies that for all $j \geq N$ we have $f(b_j) < F(b_j)$.

For each $j \geq N$, let $g_j \in \mathcal{F}$ be a function such that $g_j(b_j) = F(b_j)$. Since $\mathcal{F}$ is finite, there exists some $g \in \mathcal{F}$ such that $g = g_j$ for infinitely many values j. Let $j_0 < j_1 < \cdots$ be the subsequence such that $g = g_{j_k}$ for all $k \in \mathbb{N}$. Then for all j_k, we have $F(b_{j_k}) = g(b_{j_k})$. Further, since F and g are continuous, we have

$$g(c) = \lim_{k\to\infty} g(b_{j_k}) = \lim_{k\to\infty} F(b_{j_k}) = F(c) = f(c).$$

By (A.5), we therefore have that for all k

$$\frac{g(b_{j_k}) - g(c)}{b_{j_k} - c} = \frac{F(b_j) - F(c)}{b_j - c} \geq r + \varepsilon.$$

However, this final expression contradicts the assumption that $g'(c) \leq r$. Therefore, (A.1) holds, as desired. □