7 Network Synchronization

Chapter Contents

Learning Goals

The main goal of this chapter is to introduce the *timed message passing model* TMP, which is a more realistic middle ground between the extremes of SMP and AMP from the previous chapter. In this model, we introduce the clock synchronization problem, variants of which we will study intensively in later chapters. Moreover, we consider the notion of crash faults, where nodes fail "neatly" by stopping to execute their FSMs and sending messages. We show how to adapt the simulation technique from the previous chapter to handle crash faults in TMP, which is impossible in AMP.

7.1 Overview

In Chapter 6, the SMP and AMP models were introduced. They represent unrealistically optimistic and pessimistic extremes, respectively: SMP provides such perfect guarantees on timing that time is abstracted to the point where it is reduced to the round number, while AMP makes no assumptions on how long any operation takes whatsoever. AMP has the advantage that it puts comparatively little load on the designers of the system on which AMP algorithms run. In contrast, SMP is much more convenient for the algorithm designer, but ensuring that the system follows SMP can be highly challenging.

Chapter 6 presented a way of getting the best of both worlds. By *simulating* SMP in AMP (cf. Theorem 6.18), the advantages of SMP are made available to algorithm designers without putting the burden of directly implementing SMP on hardware developers. The latter is crucial when solutions from earlier chapters based on clock trees and PLLs become insufficient to maintain (the illusion of) perfect synchrony. However, there are situations in which this technique is insufficient. First, Theorem 6.18 provides no guarantees on relative timing between events of the same round at different nodes. Even when imposing bounds on delays, the resulting guarantees are weak: if the maximum delay is one time unit, there can be up to D time units between the times when different nodes in the network execute the same round locally; here D is the *diameter* of the graph G, which is defined to be the maximum length of a shortest path in G. This may result in poor performance when the system needs to coordinate actions in time, either internally or in relation to external events.

Second, even the most benign faults, nodes neatly "crashing" without sending any spurious messages or us requiring for them to recover later, breaks the simulation of SMP in AMP.

Definition 7.1 (Crash Faults)**.** *A network node $v \in V$* crashes *by stopping to execute its algorithm, i.e., its FSM stops performing computations and sending messages. Nodes might crash during an operation, such that only a subset of the resulting messages are sent. In SMP, this means that if node v crashes in round $r \in \mathbb{N}$, it sends an arbitrary subset of the messages it would usually send, and all other messages in this round and those sent in future rounds are replaced by $\bot$. Likewise, in AMP and TMP (see Section 7.2), if v crashes when processing an event, it sends an arbitrary subset of the messages its FSM would send due to the event, and will not respond to any future events. Similarly, local outputs may or not be generated when the node crashes, but not afterwards (in SMP, we can represent this by the special output symbol $\bot$). A node that crashes during execution $\mathcal{E}$ is referred to as being* faulty *in $\mathcal{E}$, while nodes that are not faulty in $\mathcal{E}$ are deemed* correct *in $\mathcal{E}$.*

Theorem 7.4. *Suppose for SMP algorithm $\mathcal{A}$ there are two executions in which all local inputs are identical, but (due to a crash) a local output of some $v \in V$ is different in the two executions. Then $\mathcal{A}$ cannot be simulated by an AMP algorithm.*

The *timed message passing model*, TMP, overcomes the above impossibility by adding minimal assumptions on timing to the AMP model. First, we require a *known* upper bound d on the maximum end-to-end delay; this is different from AMP, where we introduced such a bound to compare the performance of algorithms, but required the algorithm to work correctly regardless of how large delays become. Unfortunately, this, by itself, is of no use to algorithms, so long nodes cannot *measure* time. This motivates the second assumption, which is that each node is equipped with its own *hardware clock*. This clock is not perfectly accurate, but progresses at a rate between 1 and $\vartheta > 1$. Nodes are capable of reading their local clocks, which allows them to trigger a local event when their clock proves that $2d$ time has passed after an event (i.e., $2\vartheta d$ time according to their local clock elapsed).

For example, node $v \in V$ can exploit this to send a query to a neighbor w determining whether it crashed—if w does not respond with an "I'm alive" message before the above (local!) timeout has passed, the querying node can infer that w must have crashed. This simple *failure detector* allows us to modify the simulation of SMP in AMP from Theorem 6.18 to also work in TMP with crash faults. As detecting a fault takes $O(d)$ time and the simulation is held up until the problem is noticed, the time complexity of the simulation increases by $O(df)$ when facing f faults.

Theorem 7.5. *Given any algorithm $\mathcal{B}$ in SMP with crash faults, there is an algorithm $\mathcal{A}$ that simulates $\mathcal{B}$ in TMP with crash faults, in the sense that Definition 6.17 is satisfied and the same set of nodes are faulty. Simulation of the first T rounds of $\mathcal{B}$ is complete by time $(t_0 + T + O(f))d$ in executions where all nodes $v \in V$ have woken up by time $t_0 d$, for all $i \in \mathbb{N}_{>0}$ the event of v receiving $\gamma^{\mathcal{B}}_{v,i}$ happens by time $(t_0 + i)d$, and there are at most f faults. In graphs that have diameter D after removing all faulty nodes, this implies that the first T rounds of $\mathcal{B}$ are simulated by time $(D + T + O(f))d$.*

Coming back to the first issue, in this chapter we study the task of *clock synchronization,* in which the goal is to maintain at each node a *logical clock* that is well-synchronized to the logical clocks of other nodes.

Definition 7.2 (Clock Synchronization)**.** *The* clock synchronization problem *requires each node $v \in V$ to provide a subroutine* getL() *for querying its* logical clock $L_v : \mathbb{R}^+_0 \to \mathbb{R}^+_0$*, i.e., $L_v(t)$ is defined to be the result of* getL() *called at* *time t (after all operations of a possible event at time t have been completed). The goal is to minimize the* global skew

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

over all executions $\mathcal{E}$, where

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

is the global skew at time t. *The suprema and maxima above are also taken over all possible executions.*

We first tackle this task by a simple algorithm. Node $v \in V$ increase its logical clock L_v at the speed $\frac{dH}{dt}$ of its hardware clock H_v. To maintain synchronization, it will communicate when its logical clock reaches a threshold of kT for $k \in \mathbb{N}_{>0}$ and a parameter T, and set its logical clock to any received value that exceeds its current logical clock value. We show that this algorithm achieves the following skew bound.

Theorem 7.7. *Set $H \coloneqq \max_{v\in V}\{H_v(0)\} - \min_{v\in V}\{H_v(0)\}$. Then Algorithm 3 achieves*

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

We then show that a worst-case clock skew of $\Omega(uD)$ is unavoidable if end-to-end delays are between $d-u$ and d, and clocks are required to increase at an amortized constant rate.

Definition 7.9. *[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* $\min_{v\in V}\{L_v(t)\} \ge \alpha t - C$ *for all $t \in \mathbb{R}_0^+$ and all executions.*

Note that one might choose $u = d$, leading to a lower bound of $\Omega(dD)$. This is a refinement of the model taking into account that the system might provide more accurate guarantees on end-to-end delays than just an upper bound. The amortized progress condition, which stipulates that clocks actually increase, is simply necessary to enforce non-trivial solutions to the clock synchronization problem; otherwise, all nodes could return 0 as response to getL() at all times.

Theorem 7.11. *If an algorithm satisfies the amortized α-progress condition for some $\alpha \in \mathbb{R}^+$, then $\mathcal{G} \ge \frac{\alpha u D}{2}$, even if we are guaranteed that $H_v(0) = 0$ for all $v \in V$.*

Neglecting implementation issues, T can be chosen arbitrarily small. Hence, for the sake of comparing the upper and lower bounds, let us pretend that $T = 0$.

An unsatisfying gap of $(d - u/2)D + (\vartheta - 1)dD$ remains. Usually, $u \ll d$, i.e., the uncertainty about message transit and processing times can be kept much smaller than the end-to-end delay. We modify the algorithm giving rise to Theorem 7.7 to account for the fact that we know that the minimum time a message is under way is $d - u$, by adding this difference to any received value. We end up with the following improved upper bound.

Theorem 7.14. *Set $H := \max_{v \in V}\{H_v(0)\} - \min_{v \in V}\{H_v(0)\}$. Then Algorithm 4 achieves*

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

This reduces the gap to $uD/2 + (\vartheta - 1)dD$ (still pretending that $T = 0$). The first part can be explained by noting that our algorithms satisfy the more strict progress condition that $\min_{w \in V}\{H_w(t)\} \leq L_v(t) \leq \max_{w \in V}\{H_w(t)\}$ for each $v \in V$. For this natural *strong envelope condition,* a stronger lower bound of uD holds.

Theorem 7.16. *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 $v \in V$.*

This leaves us with the so far unexplained term of $(\vartheta - 1)dD$, which is due to nodes' clocks drifting apart during the up to dD time message propagation between nodes in distance D might take. However, if this term dominates, this entails that $(\vartheta - 1)d > u$. In this case, we are better off by relying on messages to compute more accurate "hardware" clocks!

E7.1 Show that, up to an additive error of $O((\vartheta - 1)d)$, a node can generate a local clock with rates from $[1, 1 + u/(d - u)]$ by playing "message ping pong" with a neighbor.

Corollary 7.3. *Set $H := \max_{v \in V}\{H_v(0)\} - \min_{v \in V}\{H_v(0)\}$ and assume that $u \leq d/2$. For any $\varepsilon > 0$, there is an algorithm that achieves*

$$\mathcal{G} \leq \max\{H, uD\} + (2u + \varepsilon)D + O((\vartheta - 1)d).$$

While this admittedly again leaves a constant factor gap to the lower bound, we can see that the obstacle is the quality of the hardware clocks. As typically $u \gg (\vartheta - 1)d$, the above results provide a fairly detailed picture of how well the clock synchronization problem can be solved.

7.2 The Timed Message Passing Model

In the timed message passing model TMP, each node v is equipped with a *hardware clock*, denoted H_v. The goal of clock synchronization is for each node to compute a *logical clock*, denoted L_v, such that all logical clocks remain as closely synchronized as possible. The challenge arises from TMP—in contrast to SMP—modeling inherent timing uncertainty in the system. Specifically, the model has two forms of uncertainty: uncertainty in the hardware clock rate, called *clock drift*, and uncertainty in the transit times of messages between nodes.

There are several possible ways of augmenting the AMP model in order to provide the node's FSM with access to its hardware clock, for example:

- add a port presenting the current hardware clock value to the FSM on each event
- model the hardware clock as a separate state machine that is queried by the node (possibly with some delay and/or inaccuracy in responses)
- provide the FSM with the ability to start timeouts (via local outputs), where a timeout of duration T being started results in an expiration event between T/ϑ and T time later

Each of these choices has different advantages and disadvantages.

The first choice provides the most power to the algorithm, but is typically furthest from (an efficient) implementation. This is made worse, as for analysis purposes we will model H_v as a continuous, strictly increasing, real-valued function, but digital logic operates with discrete values. This entails that any physical implementation must involve some form of discretization error—either because the clock is only accessed at discrete times or its readings are themselves discretized—as well as modulo operation.

The second choice allows for a more realistic modeling, but doing so would complicate the model by introducing further variability and parameters. This, in turn, makes it more cumbersome when analyzing algorithms.

Lastly, timeouts make explicit what the algorithm is keeping track of and map to implementations in a fairly intuitive manner. However, they can result in fairly unintuive descriptions of some algorithms.

Regardless of the option picked, good implementations involve careful design and low-level analysis of the components that are critical for timing. Reflecting this in the model in detail gets in the way of proving correctness and guarantees of the higher-level algorithms that are introduced in this and later chapters. The most suitable compromise is absorb these details in the uncertainty parameters of the TMP model, i.e., uncertainty in message transit times and clock drift. When implementing an algorithm, one then optimizes the hardware implementation of individual nodes and communication links to emulate the abstract behavior assumed by the TMP model in a way minimizing its uncertainty parameters.

In line with this reasoning, we will not fix in the TMP model how algorithms are mapped to FSMs, opting for a more abstract description. In pseudocode, nodes will make use of hardware clock readings by calling getH(), which returns $H_v(t)$ when called at time t. In later chapters, we will also make use of timeouts. This comes with the understanding that additional work is necessary to derive suitable hardware implementations. Nonetheless, it is instructive to think of TMP as "AMP with additional timing constraints." For every execution, we assume that there is an *objective* "true" Newtonian time taking values in $\mathbb{R}_0^+$. We will typically denote objective times by the variables t or t', referring to them simply as "time t" or "time t'," respectively. Objective time allows us to define (and reason about) the global state of the system at any given instant, but the objective time is never known to any node.

For each node $v \in V$, we model v's hardware clock as a strictly increasing function $H_v \colon \mathbb{R}_0^+ \to \mathbb{R}_0^+$. We assume that H_v increases at a rate between 1 and $\vartheta > 1$:

$$\forall v \in V, t, t' \in \mathbb{R}_0^+, t \geq t' \colon t - t' \leq H_v(t) - H_v(t') \leq \vartheta(t - t'), \tag{7.1}$$

where $t, t' \in \mathbb{R}_0^+$ denote objective times. For simplicity, we assume that hardware clocks are differentiable and denote the derivative by $\frac{dH_v}{dt}$.[6] We call $H_v(t)$ the *local time* and $\frac{dH_v}{dt}(t)$ the (instantaneous) *hardware clock rate* of v at objective time t. Observe that Equation (7.1) implies that $\frac{dL_v}{dt}(t) \in [1, \vartheta]$ at all times t. The parameter ϑ—an upper bound on the rates of all hardware clocks—is known to the algorithm designer, however nodes have no way of learning the values of $\frac{dH_v}{dt}(t)$ directly. Thus any possible (differentiable) hardware clock values H_v satisfying (7.1) are admissible, and a good clock synchronization algorithm should maintain synchronization for all possible H_v without knowledge of the rates $\frac{dH_v}{dt}$ beyond what is implied by (7.1). We 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*.[7]

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 it will take for v's message

[6] All of the claims we make can be derived from (7.1) without the assumption of differentiability, but this assumption simplifies our analysis.

[7] A cheap quartz oscillator has a drift of $\vartheta - 1 \approx 10^{-5}$, which will be more than accurate enough for running most algorithms discussed in this book. 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.

to be delivered to w. A message sent at objective time t is received at a time $t' \in (t+d-u, t+d)$, where d is the (maximum end-to-end) *delay* and u is the (delay) *uncertainty*. The delay d subsumes delays due to local computations, etc. That is, in our model, at the time t' when the message is received, all updates to the state of the receiving node take effect immediately, and messages it sends in response are also sent immediately.

An *event* consists of (1) a node starting execution of the algorithm (at the latest when receiving its first message), (2) a node sending or receiving a message, (3) a node's hardware clock reaching some prescribed value (possibly determined in response to a previous event), or (4) a node receiving a local input. Every event e seen by a node v has both an associated objective time t_e when the event occurs, and an associated local time $H_v(t_e)$ when v witnesses the event.

Regardless of how we choose to implement an algorithm, this entails that the state of a node at time t is a function of the history of events witnessed by v up to time t along with the associated local (i.e., hardware clock) times at which v witnessed the events, as well as the current hardware time $H_v(t)$. Informally, an *algorithm* specifies when and how a node responds to each event it sees, given its current state when the event occurs. We assume that an algorithm produces (and hence witnesses) a finite number of events in every bounded interval of time, but we do not make any other assumptions about nodes' local computations. This being said, an algorithm designer is well-advised to keep in mind what is practical to implement; processing messages at a high frequency might be challenging or even impossible, and is likely to affect d or u.

An *execution* of an algorithm on a system specifies hardware clock functions H_v as above for each $v \in V$, and assigns to each event e an objective time t_e at which the event occurs. In particular, a message sent by v at objective time t must be received at time $t' \in [t+d-u, t+d]$. Since an algorithm only produces finitely many events in any bounded interval of time, there is an increasing sequence of times $t_1 < t_2 < t_3 < \cdots$ at which some event(s) occur (at any node). Further, the state of the system at these times is defined inductively: given the execution and states of nodes at time t_i for $i \geq 1$, one can determine the time t_{i+1} at which the next event(s) occur, as well as the state of the system at this time.

As mentioned in Definition 7.1, we model crash faults by the failing node stopping to respond to events. If the node crashes while processing an event, it sends an arbitrary subset of the messages (which might be none) it would send if it stayed operational.

We stress that this fault model is very optimistic, in that a crashing node is required to ensure that it halts without sending out any ambiguous signals.

Error detection techniques for sanitizing communication can help in making this assumption more realistic, but are cumbersome at best in low-level hardware implementations. Therefore, we caution to view the results on crash faults in this and subsequent chapters mainly as an indication of the capabilities and limitations of the presented techniques. In later chapters, we will see that TMP allows us to handle much more general fault types, showing that it is also a practical model for designing fault-tolerant algorithms.

7.2.1 Simulating SMP with Crash Faults

As a proof of concept for TMP, we can now show that in AMP, we cannot simulate SMP with crash faults, but in TMP we can.

Intuitively, in the AMP model one should not expect to be able to handle crashes, since there is no way to distinguish between a node that has crashed and one that is just very slow to respond. This means that a simulation algorithm for SMP cannot commit to an output that depends on the content of a message that might be sent if a node has not crashed; it must wait forever to see whether the message might yet arrive.

Theorem 7.4. *Suppose for SMP algorithm $\mathcal{A}$ there are two executions in which all local inputs are identical, but (due to a crash) a local output of some $v \in V$ is different in the two executions. Then $\mathcal{A}$ cannot be simulated by an AMP algorithm.*

In the SMP model, nodes can easily detect a crash of a neighboring node if an expected message is not received. In fact, one could always require that messages are sent, meaning that crashes are always detected by all neighbors either in the round they occur or one round later. Algorithms in the SMP model then can be designed to respond to crashes in a suitable manner. In order to simulate such an algorithm, the simulation thus must also be able to detect crashes.

In TMP, we can make use of the timing guarantees to detect crashes. While this is not as trivial as in the SMP model, the idea is the same: observe that an expected message does not arrive in time. To this end, we "wrap" each link into a lower level state machine that collects messages from the link and forwards them to the port of the high level state machine responsible for the simulation. Each link state machine notifies the other endpoint of the link that the node has not crashed (yet) by sending an "empty" message (distinct from the algorithm's messages) every d time. Simultaneously, it keeps track of when the most recent message arrived from the other end. If no message arrives within $2d$ time, it determines that the other end crashed and notifies the high level state machine of the crash (again indicated by a symbol distinct from the messages of the simulation algorithm).

Notice that in order to ensure that indeed $2d$ time passed, due to the uncertainty in the hardware clock rate, the lower level state machine must wait for $2\vartheta d$ local time before declaring the other endpoint of the link crashed.

Theorem 7.5. *Given any algorithm $\mathcal{B}$ in SMP with crash faults, there is an algorithm $\mathcal{A}$ that simulates $\mathcal{B}$ in TMP with crash faults, in the sense that Definition 6.17 is satisfied and the same set of nodes are faulty. Simulation of the first T rounds of $\mathcal{B}$ is complete by time $(t_0 + T + O(f))d$ in executions where all nodes $v \in V$ have woken up by time $t_0 d$, for all $i \in \mathbb{N}_{>0}$ the event of v receiving $\gamma^{\mathcal{B}}_{v,i}$ happens by time $(t_0 + i)d$, and there are at most f faults. In graphs that have diameter D after removing all faulty nodes, this implies that the first T rounds of $\mathcal{B}$ are simulated by time $(D + T + O(f))d$.*

E7.2 Show that the additional assumptions made for TMP are minimal to overcome the impossibility shown in Theorem 7.4, i.e., if one discards either the assumption of hardware clocks or bounded message delays, the theorem applies again.

7.3 The Max Algorithm

We now move on to solving the clock synchronization problem from Definition 7.2, proving Theorem 7.7. In the following, we simplify the presentation by making an unrealistic simplification: all nodes wake up at time 0. This is of course highly impractical, as it would require perfect synchronization—and we just set out to solve this problem! However, the details of how nodes wake up are of no great importance, as any real system can simply wait for, say, a microsecond when booting, until the logical clocks have settled to guaranteeing a skew bound that is not affected by the initilialization conditions.

Moreover, in the following D denotes the diameter of the network graph *after* removing all faulty nodes. Thus, any two correct nodes are connected by a path of at most length D that does not contain any faulty nodes. Note that if the graph becomes disconnected by crashes, we cannot maintain synchronization between the different connected components any more. In this case, $D = \infty$ and the stated skew bound becomes ∞, too.[8] Moreover, nodes have no knowledge of D and the algorithm makes no use of it; it only appears in skew bounds. Hence, w.l.o.g. we assume that $D < \infty$ in the following.

[8] However, Theorem 7.7 still applies to each connectivity component when we replace D with the diameter of the component.

The algorithm is 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 3 for the pseudocode.

Algorithm 3 Basic Max Algorithm. Parameter $T \in \mathbb{R}^+$ controls how frequently messages are sent. The code lists the actions of node v at time t and provides getL().

```
 1: if $t = 0$ (i.e., $v$ just woke up) then
 2:     $h \leftarrow$ getH()
 3:     $\ell \leftarrow h$                                ▷ initialize $L_v(0)$ to $H_v(0)$
 4: end if
 5: if received $\langle \ell' \rangle$ at time $t$ and $\ell' >$ getL() then
 6:     $h \leftarrow$ getH()
 7:     $\ell \leftarrow \ell'$                  ▷ increase logical clock to received value
 8: end if
 9: if getL() $= kT$ for some $k \in \mathbb{N}$ then
10:     send $\langle kT \rangle$ to all neighbors
11: end if
12: procedure getL()                                   ▷ returns $L_v(t)$
13:     return $\ell$ + getH() $- h$          ▷ logical clock increases at rate $\frac{dH_v}{dt}$
14: end procedure
```

Lemma 7.6. *In a system executing Algorithm 3, it holds that*

$$\mathcal{G}(t) \leq \vartheta dD + (\vartheta - 1)T;$$

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

Proof. For any time t, let $L_{\max}(t) = \max_{w \in V}\{L_w(t)\}$ be the maximum logical clock value in the system at time t. Observe that any node v satisfying $L_v(t) = L_{\max}(t)$ cannot satisfy the condition in Line 5. Therefore, $L_{\max}(t)$ increases at a rate of at most ϑ (the maximum rate of any hardware clock), so that

$$L_{\max}(t') \leq L_{\max}(t) + \vartheta \cdot (t' - t) \quad \text{for all } t' > t. \tag{7.2}$$

Fix a time $t' \geq dD + T$, and let v be the node with the maximum logical clock value at time $s := t' - dD - T$. That is, $L_v(s) = L_{\max}(s)$. Applying (7.2), we find that

$$L_{\max}(t') \leq L_{\max}(s) + \vartheta \cdot (t' - s) = L_{\max}(s) + \vartheta \cdot (dD - T). \tag{7.3}$$

To finish the proof, it suffices to show that at time t', all nodes $w \in V$ satisfy $L_w(t') \geq L_{\max}(s)$. To this end observe that v's logical clock increases at a rate of at least 1, so there exists a time $s' \in [s, s+T]$ such that $L_v(s') = kT \geq L_v(s)$ for some integer $k \in \mathbb{N}$.[9] At time s', v sends the message $\langle kT \rangle$ to all of its neighbors in accordance with line 10. This message is received by all of v's neighbors by time $s' + d$, hence by time $s' + d$, all of v's neighbors' logical clocks are at least $L_{\max}(s') \geq L_{\max}(s)$. Continuing in this way, a straightforward induction argument shows that for all $\ell \in \mathbb{N}$ and all nodes w within distance ℓ from v will satisfy $L_w(s' + \ell \cdot d) \geq L_{\max}(s)$. In particular, taking $\ell = D$ (the network diameter), we find that for all $w \in V$

$$L_{\max}(s) \leq L_{\max}(s') \leq L_w(s' + D \cdot d) \leq L_w(t');$$

which implies the desired result. □

Theorem 7.7. *Set* $H := \max_{v \in V}\{H_v(0)\} - \min_{v \in V}\{H_v(0)\}$. *Then Algorithm 3 achieves*

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

Proof. Consider $t \in \mathbb{R}_0^+$. If $t \geq dD + T$, then $\mathcal{G}(t) \leq \vartheta dD + (\vartheta - 1)T$ by Lemma 7.6. If $t < dD + T$, then for any $v, w \in V$ we have that

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

7.4 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). To this end, we will use the *shifting technique,* which enables to "hide" skew from the nodes. That is, we construct two executions that 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 at least at the rate of slowest hardware clock among all nodes. First, we need to state what it means that two executions are *indistinguishable* at a node.

[9] It could be the case that L_v reaches kT because v received a message from some other node v' that overtook v as the fastest node in the network. However, our argument only relies on the fact that L_v reaches an integer multiple $kT \geq L_v(s)$ at some time in the interval $[s, s+T]$.

Definition 7.8 (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.*

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).

This has an important consequence. 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 local 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 7.9. *[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 7.10. *Fix some nodes $v, w \in V$ and $\rho \in (1, \vartheta)$ such that $(\rho - 1)d < u/2$, and set $t_0 := d(v, w)(u/(2(\rho - 1)) - d)$. For any algorithm, there are indistinguishable executions $\mathcal{E}_1$ and $\mathcal{E}_v$ satisfying that*

- $H_x^{(\mathcal{E}_1)}(t) = t$ *for all $x \in V$ and t,*
- $H_v^{(\mathcal{E}_v)}(t) = H_v^{(\mathcal{E}_1)}(t) + d(v, w)(u/2 - (\rho - 1)d)$ *for all $t \geq t_0$,*
- $H_w^{(\mathcal{E}_v)}(t) = t$ *for all t, and*
- $\mathcal{E}_1$ *does not depend on the choice of v and w.*

Proof. 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 $1 + (\rho - 1)(d(x) + d(v,w))/(2d(v,w))$ at all times $t \leq t_0$ and to 1 at all times $t > t_0$. This implies that

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

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 and 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, fix a time t and set $\lambda := \max\{t/t_0, 1\}$. We compute

$$\begin{aligned} H_x^{(\mathcal{E}_v)}(t) - H_y^{(\mathcal{E}_v)}(t) &= \frac{d(y) - d(x)}{2d(v,w)} \cdot (\rho - 1)\lambda t_0 \\ &= \lambda \cdot \frac{d(y) - d(x)}{2}\left(\frac{u}{2} - (\rho - 1)d\right). \end{aligned}$$

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 $H_y^{(\mathcal{E}_1)}(t + d - u/2) = H_x^{(\mathcal{E}_1)}(t) + d - u/2$. Thus, showing that $H_y^{(\mathcal{E}_v)}(t + d - u) < H_x^{(\mathcal{E}_v)}(t) + d - u/2 < H_x^{(\mathcal{E}_v)}(t) + d$ will complete the proof. Recall that ρ is such that $u/2 - (\rho - 1)d > 0$. We have that

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

where the second to last inequality uses that $d(x) - d(y) \geq -2$ and $0 \leq \lambda \leq 1$. On the other hand,

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

where the second inequality uses that $d(x) - d(y) \leq 2$ and $0 \leq \lambda \leq 1$. □

Theorem 7.11. *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. Fix $v, w \in V$ such that $d(v,w) = D$ and set $\rho \in (1, \vartheta)$ such that $(\rho - 1)d < u/2$. In the following, we abbreviate $\varepsilon := (\rho - 1)d$; note that we can choose $\varepsilon > 0$ arbitrarily small by picking ρ accordingly. We apply Lemma 7.10 twice, where the second time we reverse the roles of v and w. As $\mathcal{E}_1$ does not depend on the choice of v and w and indistinguishability of executions is transitive, we get two indistinguishable executions $\mathcal{E}_v$ and $\mathcal{E}_w$ such that there is a time t_0 satisfying for all $t \geq t_0$ that

- $H_v^{(\mathcal{E}_w)}(t) = H_w^{(\mathcal{E}_v)}(t) = t$ and
- $H_v^{(\mathcal{E}_v)}(t) = H_w^{(\mathcal{E}_w)}(t) = t + (u/2 - \varepsilon)D$.

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

$$L_v^{(\mathcal{E}_w)}\left(t + \left(\frac{u}{2} - \varepsilon\right)D\right) - L_v^{(\mathcal{E}_w)}(t) + L_w^{(\mathcal{E}_v)}\left(t + \left(\frac{u}{2} - \varepsilon\right)D\right) - L_w^{(\mathcal{E}_v)}(t) \\ \geq \alpha(u - 3\varepsilon)D. \tag{7.4}$$

Assuming for contradiction that this is false, set $0 < 2\alpha' := \frac{\alpha(u-3\varepsilon)D}{(u/2-\varepsilon)D} < 2\alpha$ and consider times $t_k := t_0 + k(u/2-\varepsilon)D$ for $k \in \mathbb{N}$. By induction over k, we get that

$$\begin{aligned} L_v^{(\mathcal{E}_w)}(t_k) + L_w^{(\mathcal{E}_v)}(t_k) &\le L_v^{(\mathcal{E}_w)}(t_0) + L_w^{(\mathcal{E}_v)}(t_0) + 2\alpha'(t_k - t_0) \\ &\le 2\alpha t_k - 2(\alpha-\alpha')t_k + L_v^{(\mathcal{E}_w)}(t_0) + L_w^{(\mathcal{E}_v)}(t_0). \end{aligned}$$

Choosing k large enough so that $t_k > (L_v^{(\mathcal{E}_w)}(t_0) + L_w^{(\mathcal{E}_v)}(t_0) + 2C)/(2(\alpha-\alpha'))$, we get that

$$L_v^{(\mathcal{E}_w)}(t_k) + L_w^{(\mathcal{E}_v)}(t_k) < 2(\alpha t_k - C).$$

Therefore, $L_v^{(\mathcal{E}_w)}(t_k) < \alpha t_k - C$ or $L_w^{(\mathcal{E}_v)}(t_k) < \alpha t_k - C$, violating the α-progress condition in at least one of the executions. This is a contradiction, i.e., the claim must hold true.

Now let $t \ge t_0$ be such that (7.4) holds. As $H_v^{(\mathcal{E}_w)}(t+(u/2-\varepsilon)D) = t+(u/2-\varepsilon)D = H_v^{(\mathcal{E}_v)}(t)$, by indistinguishability of $\mathcal{E}_v$ and $\mathcal{E}_w$ we have that $L_v^{(\mathcal{E}_v)}(t) = L_v^{(\mathcal{E}_w)}(t+(u/2-\varepsilon)D)$. Symetrically, $L_w^{(\mathcal{E}_w)}(t) = L_w^{(\mathcal{E}_v)}(t+(u/2-\varepsilon)D)$. Hence,

$$\begin{aligned} &|L_v^{(\mathcal{E}_v)}(t) - L_w^{(\mathcal{E}_v)}(t)| + |L_v^{(\mathcal{E}_w)}(t) - L_w^{(\mathcal{E}_w)}(t)| \\ \ge\ & L_v^{(\mathcal{E}_v)}(t) - L_w^{(\mathcal{E}_v)}(t) + L_w^{(\mathcal{E}_w)}(t) - L_v^{(\mathcal{E}_w)}(t) \\ =\ & L_v^{(\mathcal{E}_w)}\left(t + \left(\frac{u}{2}-\varepsilon\right)D\right) - L_w^{(\mathcal{E}_v)}(t) + L_w^{(\mathcal{E}_v)}\left(t + \left(\frac{u}{2}-\varepsilon\right)D\right) - L_v^{(\mathcal{E}_w)}(t) \\ =\ & L_v^{(\mathcal{E}_w)}\left(t + \left(\frac{u}{2}-\varepsilon\right)D\right) - L_v^{(\mathcal{E}_w)}(t) + L_w^{(\mathcal{E}_v)}\left(t + \left(\frac{u}{2}-\varepsilon\right)D\right) - L_w^{(\mathcal{E}_v)}(t) \\ \ge\ & \alpha(u-3\varepsilon)D. \end{aligned}$$

We conclude that in at least one of the two executions, the logical clock difference between v and w reaches at least $(\alpha(u-3\varepsilon)D)/2$. As $\varepsilon > 0$ can be chosen arbitrarily small, it follows that $\mathcal{G} \ge \frac{\alpha u D}{2}$, as claimed. □

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.

7.5 Refining the Max Algorithm

When propagating information, we have not factored in yet that we *know* that messages are under way for at least $d-u$ time. In order to get closer to the lower bound, we now seek to exploit this.

Lemma 7.12. *In a system executing Algorithm 4, no $v \in V$ ever sets L_v to a value larger than* $\max_{w \in V \setminus \{v\}}\{L_w(t)\}$.

Algorithm 4 Refined Max Algorithm. Note that now nodes send messages based on their hardware clock readings. This is to avoid indefinitely cascading messages, since now the received values are increased by $d - u$.

```
if $t = 0$ (i.e., $v$ just woke up) then
    $h \leftarrow$ getH()
    $\ell \leftarrow h$                                    ▷ initialize $L_v(0)$ to $H_v(0)$
end if
if received $\langle \ell' \rangle$ at time $t$ and $\ell' >$ getL() then
    $h \leftarrow$ getH()
    $\ell \leftarrow \ell' + d - u$                         ▷ increase logical clock, adding $d - u$
end if
if getH() $= kT$ for some $k \in \mathbb{N}$ then
    send $\langle$getL()$\rangle$ to all neighbors
end if
procedure getL()                                           ▷ returns $L_v(t)$
    return $\ell$ + getH() $- h$                            ▷ logical clock increases at rate $\frac{dH_v}{dt}$
end procedure
```

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 7.13. *In a system executing Algorithm 4, it holds that*

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

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

Proof. Set $L := \max_{v \in V}\{L_v(t - (d + T)D)\}$. By Lemma 7.12 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 $(v_{D-h} = v, v_{D-h+1}, \dots, v_D = w)$, where $h \le D$, a shortest v-w-path. Define $t_i := t-(D-i)(d+T)$. We prove by induction over $i \in \{D-h, D-h+1, \dots, D\}$ that

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

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

$$L_v(t_{D-h}) \ge L_v(t-(d+T)D) + t_{D-h} - (t-(d+T)D) = L + (D-h)(d+T).$$

For the induction step from $i-1 \in \{D-h, \dots, D-1\}$ 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

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

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

Theorem 7.14. *Set* $H := \max_{v\in V}\{H_v(0)\} - \min_{v\in V}\{H_v(0)\}$. *Then Algorithm 4 achieves*

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

Proof. Consider $t \in \mathbb{R}_0^+$. If $t \ge (d+T)D$, then $\mathcal{G}(t) \le uD + (\vartheta-1)(d+T)D$ by Lemma 7.13. If $t < (d+T)D$, then for any $v, w \in V$ we have that

$$L_v(t) - L_w(t) \le L_v(0) - L_w(0) + (\vartheta-1)t \le H + (\vartheta-1)(d+T)D. \quad \square$$

A few 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 $\mathcal{G} \in \max\{H, uD\} + O((\vartheta - 1)dD)$. 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 building 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.
- It is possible to achieve a skew of $O(uD + (\vartheta - 1)d)$.
- 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. Hence, changing the network topology has no effect beyond influencing the diameter.

7.6 Afterthought: Stronger Lower Bound

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

Definition 7.15 (Strong Envelope Condition)**.** *An algorithm fulfills the* strong envelope condition, *if at all times and for all nodes $v \in V$, it holds that* $\min_{w \in V}\{H_w(t)\} \leq L_v(t) \leq \max_{w \in V}\{H_w(t)\}$.

For this stronger condition, one can show that the term of uD in the skew bound is optimal.

Theorem 7.16. *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 sketch. It is possible to adapt Lemma 7.10 such that the execution $\mathcal{E}_1$ is not using delays of roughly $u/2$ between any pair of nodes, but delays are roughly $d - u$ when messages are sent "in direction of w" and d when they are sent "in direction of v." This is very similar to the use of $d(x)$ in $\mathcal{E}_v$, but we use the uncertainty "the other way round." This implies that the hardware clock difference at v between $\mathcal{E}_1$ and $\mathcal{E}_v$ can be increased to about uD (as opposed to only $uD/2$) before we run out of slack in the delays. However, in $\mathcal{E}_1$ still $H_x(t) = t$ for all $x \in V$ and times t, so nodes must maintain that $H_x(t) = L_x(t)$ in $\mathcal{E}_1$ to satisfy the strong envelope condition. Because $\mathcal{E}_v$ is indistinguishable from $\mathcal{E}_1$, the same is true in $\mathcal{E}_v$. In particular,

$$L_v^{\mathcal{E}_v}(t_0) - L_w^{\mathcal{E}_v}(t_0) \approx L_v^{\mathcal{E}_1}(t_0 + uD) - L_w^{\mathcal{E}_1}(t_0) = uD. \qquad \square$$

We remark that if one merely requires the weaker progress condition $t \leq L_v(t) \leq \max_{v \in V}\{H_v(0)\} + \vartheta t$, then a lower bound of $\frac{uD}{\vartheta}$ can be shown.

Bibliography