Before You Start Reading. . .

. . . here is some context for the material that follows. This course follows part of a book, which is currently a work in progress. Please expect and forgive minor flaws in the presentation. There will be a lot of references to previous chapters of the book. We do *not* require or expect you to study these earlier chapters. Their contents are not needed to follow the material of this course. However, they describe building blocks whose existence we take for granted for the purpose of this lecture, providing the tools to implement the presented algorithms directly in hardware. They also sketch the state of the art in terms of clocking synchronous hardware systems, ultimately motivating to consider distributed and fault-tolerant clocking methods for modern and future systems. Studying such clocking methods is the goal of the present course. Feel free to browse through the respective material for context, which you can find at https://www.mpi-inf.mpg.de/departments/algorithms-complexity/teaching/winter20/how-to-clock-your-computer, but rest assured that this is purely optional.

Given that the book-in-progress, which will provide you with the reading material covering the contents of the course, *does* assume some knowledge of previous chapters, you will require some additional context to start reading. We provide this context here in compact form. Before doing so, we quickly summarize why the introduced notions are of interest.

Motivation

In this course, we will focus on distributed fault-tolerant clock generation and distribution methods. This is motivated by a need for scalable, reliable, and high-precision synchronization across clocked hardware systems, like the CPU of laptop or smartphone. In such synchronous systems, computations are executed in a lock-step fashion, where computational steps and communication of (intermediate) results alternate. Ultimately, this allows us to implement (synchronous) state machines in an efficient way, which in turn let us build the elaborate abstractions—general purpose computers, OSes, programming languages and compilers, etc.—fuelling the incredibly powerful computing platforms we nowadays can field in our pockets.

This mode of operation relies on a precisely timed high-frequency clock signal available across the entire device. We abstract the task of making this signal available by a generic model with only a handful of parameters. We then strive to solve it using few resources, achieving high precision, and with great robustness to both transient and permanent faults.

Model

The system is described as a (directed or undirected) network $G = (V, E)$, where nodes are computational entities and edges communication links. Nodes correspond to parts of the hardware device (e.g., an area on a chip), which they are providing with a local clock signal driving the computations in this part.[1] Since adjacent parts will communicate with each other, the communicating components need to receive their clock signals as precisely synchronized as possible, i.e., with minimal relative offset. The better the synchronization, the faster, i.e., at higher frequency, computations on the device can be performed while safely maintaining synchronous operation.

As our goal is synchronization, time is integral to our model. In our algorithms, nodes need some way of measuring time. For simplicity, we abstract away from the specific implementation by assuming that each node $v \in V$ has an (imperfect) *hardware clock*. The hardware clock of v is described as a function $H_v \colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ mapping the "true" time t, which is unknown to v, to the local time $H_v(t)$ at time t. At any time, node v can access $H_v(t)$ by calling getH(). Node v can take decisions based on the values returned by calls to getH(), but also wait until a certain local time is reached. For example, the instruction "wait until local time H" implies that the node sleeps until the time t satisfying that a call to getH() would return H. The quality of the hardware clocks is measured by the parameter $\vartheta > 1$. Each hardware clock satisfies that

$$\forall t' \geq t \colon H_v(t') - H_v(t) \leq t' - t \leq \vartheta(H_v(t') - H_v(t)),$$

i.e., hardware clocks progress at rates between 1 and ϑ relative to real time. For simplicitly, we assume that the clocks are differentiable, meaning that this assumption is equivalent to $\frac{dH_v}{dt}(t) \in [1, \vartheta]$ at all times $t \in \mathbb{R}_{\geq 0}$.

Communication is by passing messages along the network links. If a node v sends a message along edge $(v, w) \in E$ at time t_s, there is a time $t_r \in [t_s + d - u, t_s + d]$ when it has been received and fully processed by w, i.e., all immediately triggered computations are complete and w has sent any resulting messages. In other words, the *end-to-end delay* (or delay, for short) d upper bounds the sum of communication and computational delay. Note that this sum is at least $d - u$, where we refer to u as the *uncertainty* (in end-to-end delay). The parameters d and u are known, i.e., node w knows that a message it finishes to process at time t_r must have been sent at some time $t_s \in [t_r - d, t_r - d + u]$.

[1] This signal is typically locally distributed by a so-called clock tree. For the purpose of this course, we will neglect this aspect and focus on synchronizing the clock signals the nodes produce.

Task

Our goal is to produce synchronized clock signals at the nodes of the system. While there will be variants, let us specify the *clock synchronization* task to provide intuition on the kind of problems we seek to solve in this course. In this task, the goal is for each node to compute a *logical clock,* described by a function $L_v\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, to which it provides access in the form of a subroutine getL() that can be called at any time t and returns $L_v(t)$ (where, again, t is not known to the node). A desirable property of such clocks is that they run at bounded rates, i.e.,

$$\forall t' \geq t\colon L_v(t') - L_v(t) \leq t' - t \leq \beta(L_v(t') - L_v(t)),$$

for some $\beta > 1$, where similar to the hardware clocks we took the liberty to normalize the minimum rate to be 1. Again, we will for simplicity assume that the logical clocks are differentiable, i.e., this is equivalent to $L_v(t) \in [1, \beta]$ for all times t. We remark that, to avoid needless complication, algorithms might sometimes produce non-differentiable clocks. If such violations are limited to changing clock rates instantaneously at discrete points in time, we will pretend that $\frac{dL_v}{dt}$ is defined everywhere.[2]

Naturally, $L_v(t)$ depends on the (adversarially chosen) rates of the hardware clocks and delays of messages. For a given execution, specified by picking a graph G, fixing an algorithm, choosing H_v for each $v \in V$ with rates between 1 and ϑ, and assigning a delay from $[d - u, d]$ to each message the algorithm sends, we can now express what "synchronizing clocks" means formally. The quality measure of the output clocks is the *global skew at time t*

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

The algorithm should bound $\mathcal{G}(t)$ at all times, i.e., minimize

$$\mathcal{G} := \sup_{t \in \mathbb{R}_{\geq 0}} \{\mathcal{G}(t)\}.$$

While in most cases we assume a specific execution to be fixed for the purpose of our analysis, it is understood that a good algorithm does bound the skew in *all* executions. However, this (worst-case) bound will naturally depend on d, u, and ϑ, and sometimes other parameters, such as β or the diameter D (i.e., maximum distance between any pair of nodes in G) of the underlying communication network.

[2] Since $\frac{dL_v}{dt}$ only appears in integrals, this is mathematically sound. Alternatively, one could approximate L_v arbitrarily well by a differentiable function with the same bounds on its derivative.