Exercise 4: Fault Frustration

Task 1: I'm Getting Tired of these Delays!

a) Show that the Srikanth-Toueg algorithm has skew at most $d + u$.

b) Show that this skew bound is tight. It suffices to do so for the maximal number of faults $\lceil n/3 \rceil - 1$.

Solution

a) We revisit the last paragraph of the proof of Lemma 4.3. If t' is the minimum time after t_p when a node transitions to PULSE, it must have received $n-f \geq f+1$ PROPOSE messages by that time. The senders of these messages send PROPOSE messages to all nodes. These must arrive by time $t' + u$: if t_s is such a sending time, we have that $t' \geq t_s + d - u$, but all messages arrive by time $t_s + d \leq t' + u$. Thus, all non-faulty nodes have transitioned to propose by time $t' + u$ and sent respective messages. By time $t' + d + u$, all non-faulty nodes have transitioned to PULSE.

b) Split the correct nodes into sets F and S, where $|F| = n - 2f \geq f + 1$ and $|S| = n - f - |F| = f$. All nodes have hardware clock rate 1. Messages to nodes in S are delayed by $d - \varepsilon$ (for some arbitrarily small $\varepsilon > 0$), while messages to nodes in F are delayed by $d - u + \varepsilon$. $H_v(0) \coloneqq 0$ for all $v \in V_g$. Faulty nodes never send messages to nodes in S and keep sending PROPOSE messages to nodes in F that arrive exactly on each transition to START and READY.

   Observe that (i) nodes in F always transition at the same times, (ii) nodes in S always transition at the same times, (iii) all nodes always take the exact same time to transition from PULSE to READY, and, unless "pulled" by the $f+1$ threshold rule, the same time to transition from READY to PROPOSE, (iv) if the $f+1$ threshold rule applies, this is for the nodes in S and happens $d - \varepsilon$ time after the nodes from F transitioned to PROPOSE, and (v) if the $f+1$ threshold rule does not apply, the time difference between nodes in F and S transitioning to PULSE increases by $u - 2\varepsilon$.

   Hence, eventually there will be an iterations in which the nodes in F will transition to PROPOSE when the messages from nodes in F arrive, i.e., $d-\varepsilon$ time after them. It takes them another $d - \varepsilon$ time to receive the messages of their fellow nodes in S and transition to PULSE. In constrast, the nodes in F transition to PULSE already $d-u+\varepsilon$ time after transitiong to PROPOSE. We thus have a skew of $2(d - \varepsilon) - (d - u + \varepsilon) \geq d + u - 3\varepsilon$. As ε can be chosen arbitrarily small, it follows that $\mathcal{S} \geq d + u$.

Task 2: Stop Failing and Start Synchronizing!

In this exercise, $3f \geq n$, i.e., there may be "too many" Byzantine nodes.

a) Show that clock synchronization is impossible with this many faults if the constant amortized progress condition is satisfied. (Hint: First "spend" some of the uncertainty and clock drifts to show that logical clocks cannot increase too rapidly. Then argue that any solution would imply a pulse synchronization algorithm.)

b) Show that even with this many faults, there is an algorithm that achieves constant skew and has unbounded logical clocks. You may assume that $\max_{v \in V_g}\{H_v(0)\} \leq H \in \mathbb{R}^+$. (Hint: Solve the problem *without* communication!)

c) Is the solution from b) useful? (Remark: This is an open-ended discussion. There is not necessarily a single right answer.)

Solution

a) Assume for contradiction that there was an algorithm $\mathcal{A}$ with constant amortized progress that solved clock synchronization with bounded skew $\mathcal{G}$ and tolerates $f \geq n/3$ Byzantine faults. Set $u' \coloneqq u/2$, $d' \coloneqq d - u/4$, and $\vartheta' \coloneqq 1 + (\vartheta - 1)/2$. Consider the algorithm on executions with delays from $(d' - u', d')$ and hardware clock rates from $[1, \vartheta']$. Applying Lemma 2.4, we can see that no node ever can increase its logical clock by more than $2\mathcal{G}$ in $u/4$ time.

Using this information and assuming that $H_v(0) = 0$ for all $v \in V_g$, we can construct a pulse synchronization algorithm as follows. We run $\mathcal{A}$, where each node generates a pulse whenever its logical clock reaches a multiple of a (to-be-determined) parameter T that is larger than $\max_{v \in V_g}\{L_v(0)\}$; note that the latter value is known, as $L_v(0)$ is determined by $H_v(0)$. As $\mathcal{A}$ satisfies constant amortized progress, there is some $\mathcal{S}$ (depending on $\mathcal{G}$ and the constants in the progress condition) such that for each $k \in \mathbb{N}$, it holds that

$$\max\{t \in \mathbb{R}_0^+ \mid \exists v \in V_g \colon L_v(t) = kT\} - \min\{t \in \mathbb{R}_0^+ \mid \exists v \in V_g \colon L_v(t) = kT\} \leq \mathcal{S}\,,$$

i.e., the pulse synchronization algorithm has skew $\mathcal{S}$. By choosing $T \coloneqq 16\mathcal{G}\mathcal{S}/u$, the bound on the increase of logical clocks implies that it takes at least $2\mathcal{S}$ time for a node that generates a pulse to generate the next. Thus, we have that $P_{\min} \coloneqq \mathcal{S}$ is a valid minimum period for the algorithm. Lastly, because $\mathcal{A}$ satisfies the amortized progress condition, there is some bounded maximum period $P_{\max}$ that the new algorithm satisfies.

In summary, we have constructed a pulse synchronization algorithm for $3f \geq n$ under the condition that $H_v(0) = 0$ for all $v \in V$ (with modified delay bounds and clock drift, which does not matter). This is a contradiction to Theorem 4.6, so the original assumption that a clock synchronization algorithm with constant amortized progress and bounded skew tolerating $f > n/3$ faults must be wrong.

b) We set $L_v(t) \coloneqq \log(1 + H_v(t))$ for all $v \in V_g$. As the logarithm is increasing and $\log(a + b) \leq \log a + \log b$ for $a, b \geq 1$, this implies for $v, w \in V$ and $t \in \mathbb{R}_0^+$ that

$$\begin{aligned} L_v(t) - L_w(t) &= \log(1 + H_v(t)) - \log(1 + H_w(t)) \\ &\leq \log(1 + H + \vartheta t) - \log(1 + t) \\ &\leq \log(1 + H) + \log(\vartheta(1 + t)) - \log(1 + t) \\ &\leq H + \log \vartheta\,. \end{aligned}$$

c) All the "algorithm" from b) does is to compute a function of the hardware clock. This function cannot contain any additional information or useful guarantee. Concretely, here it is just rescaling clocks non-uniformly to keep the skew of the logical clocks small, but the clocks in turn cannot be used to synchronize events at the different nodes any better than directly using the hardware clock values would.