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*

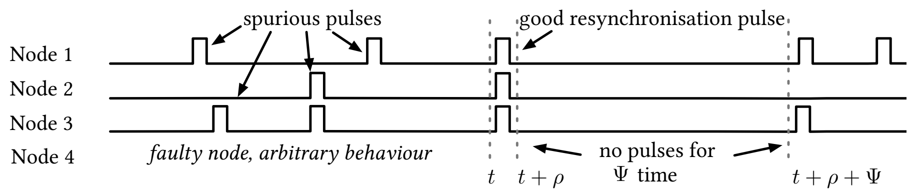
- *it stabilizes in $\mathcal{O}(df)$ time,*
- *correct nodes broadcast $\mathcal{O}(\log f)$ bits per d time,*
- *it has skew $2d$,*
- *it has minimum period $P_{\min} \geq T$, and*
- *it has maximum period $P_{\max} \leq \varphi T$.*

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

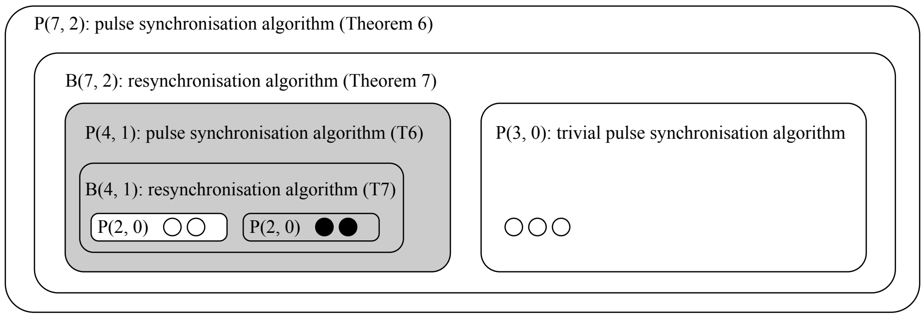

P(7, 2): pulse synchronisation algorithm (Theorem 6)
B(7, 2): resynchronisation algorithm (Theorem 7)
P(4, 1): pulse synchronisation algorithm (T6)
B(4, 1): resynchronisation algorithm (T7)
P(2, 0)
P(2, 0)
P(3, 0): trivial pulse synchronisation algorithm

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.

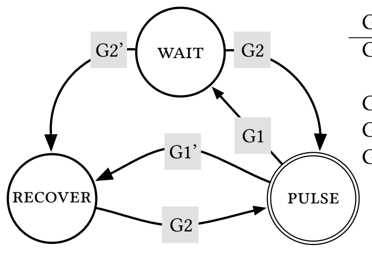
Our approach to generating pulses will be to execute consensus for each pulse. The challenge is to stabilize this procedure.

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 of 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 the PULSE state when Guard G2 is satisfied, which requires an OUTPUT 1 signal from the auxiliary state machine given in Figure 12.3.

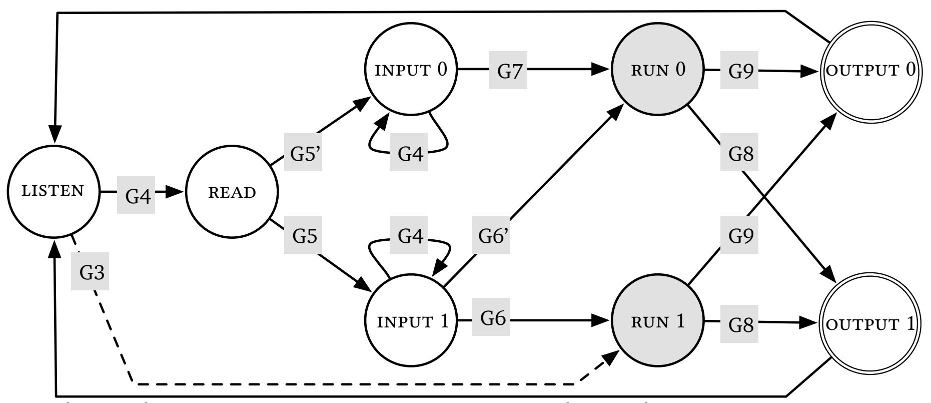
Remarks:

- 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 Figure 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 Figure 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 $\mathcal{O}(dR)$ 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 READ 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 the 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 Figure 12.2 and Figure 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 LISTEN to 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 $\log^{\mathcal{O}(1)} f$ (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) barrier 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 $\mathcal{O}(fd)$ time to $\mathcal{O}(1)$ per d time. 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 [LR17**?**] overcomes this restriction by relying on consensus.

It is worth noting that most constructions, in particular those of with stabilization time $\mathcal{O}(fd)$, 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.