Self-Stabilising Byzantine Clock Synchronisation is Almost as Easy as Consensus*

Christoph Lenzen[1] and Joel Rybicki†[2]

1 **Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany**
clenzen@mpi-inf.mpg.de

2 **Department of Biosciences, University of Helsinki, Finland**
joel.rybicki@helsinki.fi

Abstract

We give fault-tolerant algorithms for establishing synchrony in distributed systems in which each of the n nodes has its own clock. Our algorithms operate in a very strong fault model: we require self-stabilisation, i.e., the initial state of the system may be arbitrary, and there can be up to $f < n/3$ ongoing Byzantine faults, i.e., nodes that deviate from the protocol in an arbitrary manner. Furthermore, we assume that the local clocks of the nodes may progress at different speeds (clock drift) and communication has bounded delay. In this model, we study the pulse synchronisation problem, where the task is to guarantee that eventually all correct nodes generate well-separated local pulse events (i.e., unlabelled logical clock ticks) in a synchronised manner.

Compared to prior work, we achieve *exponential* improvements in stabilisation time and the number of communicated bits, and give the first sublinear-time algorithm for the problem:

- In the deterministic setting, the state-of-the-art solutions stabilise in time $\Theta(f)$ and have each node broadcast $\Theta(f \log f)$ bits per time unit. We exponentially reduce the number of bits broadcasted per time unit to $\Theta(\log f)$ while retaining the same stabilisation time.
- In the randomised setting, the state-of-the-art solutions stabilise in time $\Theta(f)$ and have each node broadcast $O(1)$ bits per time unit. We exponentially reduce the stabilisation time to $\operatorname{polylog} f$ while each node broadcasts $\operatorname{polylog} f$ bits per time unit.

These results are obtained by means of a recursive approach reducing the above task of *self-stabilising* pulse synchronisation in the *bounded-delay* model to *non-self-stabilising* binary consensus in the *synchronous* model. In general, our approach introduces at most logarithmic overheads in terms of stabilisation time and broadcasted bits over the underlying consensus routine.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Byzantine faults, self-stabilisation, clock synchronisation, consensus

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.32

1 Introduction

Many of the most fundamental problems in distributed computing relate to timing and fault tolerance. Even though most distributed systems are inherently asynchronous, it is often convenient to design such systems by assuming some degree of synchrony provided by reliable global or distributed clocks. For example, the vast majority of existing Very

* Full version available on arXiv [27], http://arxiv.org/abs/1705.06173.
† Part of this work was done while JR was affiliated with Helsinki Institute for Information Technology HIIT, Department of Computer Science, Aalto University.

Large Scale Integrated (VLSI) circuits operate according to the synchronous paradigm: an internal clock signal is distributed throughout the chip neatly controlling alternation between computation and communication steps. Of course, establishing the synchronous abstraction is of high interest in numerous other large-scale distributed systems, as it makes the design of algorithms considerably easier.

However, as the accuracy and availability of the clock signal is typically one of the most basic assumptions, clocking errors affect system behavior in unpredictable ways that are often hard – if not impossible – to tackle at higher system layers. Therefore, *reliably* generating and distributing a joint clock is an essential task in distributed systems. Unfortunately, the cost of providing fault-tolerant synchronisation and clocking is still poorly understood.

Pulse synchronisation. In this work, we study the *self-stabilising Byzantine pulse synchronisation* problem [16, 9], which requires the system to achieve synchronisation despite severe faults. We assume a fully connected message-passing system of n nodes, where

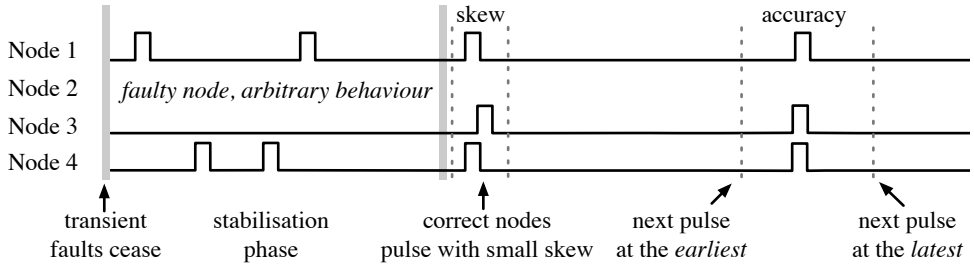
1. an unbounded number of transients faults may occur anywhere in the network, and
2. up to $f < n/3$ of the nodes can be faulty and exhibit *arbitrary* ongoing misbehaviour.

In particular, the transient faults may arbitrarily corrupt the state of the nodes and result in loss of synchrony. Moreover, the nodes that remain faulty may deviate from any given protocol, behave adversarially, and collude to disrupt the other nodes by sending them *different* misinformation even after transient faults have ceased. Note that this also covers faults of the communication network, as we may map faults of communication links to one of their respective endpoints. The goal is now to (re-)establish synchronisation once transient faults cease, despite up to $f < n/3$ Byzantine nodes. That is, we need to consider algorithms that are simultaneously (1) self-stabilising [7, 15] and (2) Byzantine fault-tolerant [23].

More specifically, the problem is as follows: after transient faults cease, no matter what is the initial state of the system, the choice of up to $f < n/3$ faulty nodes, and the behaviour of the faulty nodes, we require that after a bounded *stabilisation time* all the *non-faulty* nodes must generate pulses that

- occur almost simultaneously at each correctly operating node (i.e., have small *skew*), and
- satisfy specified minimum and maximum frequency bounds (*accuracy*).

While the system may have arbitrary behaviour during the initial stabilisation phase due to the effects of transient faults, eventually the above conditions provide synchronised unlabelled clock ticks for all non-faulty nodes:



In order to meet these requirements, it is necessary that nodes can estimate the progress of time. To this end, we assume that nodes are equipped with (continuous, real-valued) hardware clocks that run at speeds that may vary arbitrarily within 1 and ϑ, where $\vartheta \in O(1)$. That is, we normalize minimum clock speed to 1 and assume that the clocks have drift bounded by a constant. Observe that in an asynchronous system, i.e., one in which communication and/or computation may take unknown and unbounded time, even perfect clocks are insufficient to ensure any relative timing guarantees between the actions of different nodes. Therefore, we

additionally assume that the nodes can send messages to each other that are received and processed within at most $d \in \Theta(1)$ time. The clock speeds and message delays can behave adversarially within the respective bounds given by ϑ and d.

In summary, this yields a highly adversarial model of computing, where further restrictions would render the task infeasible: (1) transient faults are arbitrary and may involve the entire network; (2) ongoing faults are arbitrary, cover erroneous behavior of both the nodes and the communication links, and the problem is not solvable if $f \geq n/3$ [10]; and (3) the assumptions on the accuracy of local clocks and communication delay are minimal to guarantee solvability.

Background and related work. If one takes any one of the elements described above out of the picture, then this greatly simplifies the problem. Without permanent faults, the problem becomes trivial: it suffices to have all nodes follow a designated leader. Without transient faults [22], straightforward solutions are given by elegant classics [33, 34], where [34] also guarantees asymptotically optimal skew [29]. Taking the uncertainty of unknown message delays and drifting clocks out of the equation leads to the so-called digital clock synchronisation problem [3, 11, 25, 28, 26], where communication proceeds in synchronous rounds and the task is to agree on a consistent (bounded) round counter. While this abstraction is unrealistic as a basic system model, it yields conceptual insights into the pulse synchronisation problem in the bounded-delay model. Moreover, it is useful to assign numbers to pulses after pulse synchronisation is solved, in order to get a fully-fledged shared system-wide clock [24].

In contrast to these relaxed problem formulations, the pulse synchronisation problem was initially considered to be very challenging – if not impossible – to solve. In a seminal article, Dolev and Welch [16] proved otherwise, albeit with an algorithm having an impractical exponential stabilisation time. In a subsequent line of work, the stabilisation time was reduced to polynomial [6] and then linear in f [12]. However, the linear-time algorithm relies on simulating multiple instances of synchronous *consensus* algorithms [30] concurrently, which results in a high communication complexity.

The consensus problem [30, 23] is one of the fundamental primitives in fault-tolerant computing. Most relevant to this work is synchronous binary consensus with (up to f) Byzantine faults. Here, node v is given an input $x(v) \in \{0,1\}$, and it must output $y(v) \in \{0,1\}$ such that the following properties hold:

1. **Agreement:** There exists $y \in \{0,1\}$ such that $y(v) = y$ for all correct nodes v.
2. **Validity:** If for $x \in \{0,1\}$ it holds that $x(v) = x$ for all correct nodes v, then $y = x$.
3. **Termination:** All correct nodes eventually decide on $y(v)$ and terminate.

In this setting, two of the above main obstacles are not present: the system is properly initialised (no self-stabilisation required) and computation proceeds in synchronous rounds, i.e., well-ordered compute-send-receive cycles. This confines the task to understanding how to deal with the interference from Byzantine nodes. Synchronous consensus is extremely well-studied; see e.g. [32] for a survey. It is known that precisely $\lfloor (n-1)/3 \rfloor$ faults can be tolerated in a system of n nodes [30], $\Omega(nf)$ messages need to be sent in total [14], the connectivity of the communication network must be at least $2f+1$ [8], deterministic algorithms require $f+1$ rounds [19, 1], and randomised algorithms can solve the problem in constant expected time [18]. In constrast, no non-trivial lower bounds on the time or communication complexity of pulse synchronisation are known.

The linear-time pulse synchronisation algorithm in [12] relies on simulating (up to) one synchronous consensus instance for each node simultaneously. Accordingly, this protocol requires each node to broadcast $\Theta(f \log f)$ bits per time unit. Moreover, the use of *de-*

Table 1 Summary of pulse synchronisation algorithms for $f \in \Theta(n)$. For each respective algorithm, the first two columns give the stabilisation time and the number of bits broadcasted by a node per time unit. The third column denotes whether algorithm is deterministic or randomised. The fourth column indicates additional details or model assumptions. All algorithms tolerate $f < n/3$ faulty nodes except for (*), where we have $f < n/(3+\varepsilon)$ for any constant $\varepsilon > 0$.

time	bits	type	notes	reference
poly f	$O(\log f)$	det.		[6]
$O(f)$	$O(f \log f)$	det.		[12]
$O(f)$	$O(\log f)$	det.		this work and [4]
$2^{O(f)}$	$O(1)$	rand.	adversary cannot predict coin flips	[16]
$O(f)$	$O(1)$	rand.	adversary cannot predict coin flips	[9]
polylog f	polylog f	rand.	private channels, (*)	this work and [21]
$O(\log f)$	poly f	rand.	private channels	this work and [18]

terministic consensus is crucial, as failure of any consensus instance to generate correct output within a prespecified time bound may result in loss of synchrony, i.e., the algorithm would fail *after* apparent stabilisation. In [9], these obstacles were overcome by avoiding the use of consensus by reducing the pulse synchronisation problem to the easier task of generating at least one well-separated "resynchronisation point", which is roughly uniformly distributed within any period of $\Theta(f)$ time. This can be achieved by trying to initiate such a resynchronisation point at random times, in combination with threshold voting and locally checked timing constraints to rein in the influence of Byzantine nodes. In a way, this seems much simpler than solving consensus, but the randomisation used to obtain a suitable resynchronisation point strongly reminds of the power provided by shared coins [31, 2, 18, 3] – and this is exactly what the core routine of the expected constant-round consensus algorithm from [18] provides.

Contributions. Our main result is a framework that reduces pulse synchronisation to an arbitrary (non-self-stabilising) synchronous binary consensus routine at very small overheads. In other words, given *any* efficient algorithm that solves consensus in the standard synchronous model of computing, we show how to obtain an efficient algorithm that solves the pulse synchronisation problem in the bounded-delay model with clock drift.

While we build upon existing techniques, our approach has many key differences. First of all, while Dolev et al. [9] also utilise the concept of resynchronisation pulses, these are generated probabilistically. Moreover, their approach has an inherent time bound of $\Omega(f)$ for generating such pulses. In contrast, we devise a new recursive scheme that allows us to (1) *deterministically* generate resynchronisation pulses in $\Theta(f)$ time and (2) *probabilistically* generate resynchronisation pulses in $o(f)$ time. To construct algorithms that generate resynchronisation pulses, we employ resilience boosting and filtering techniques inspired by our recent line of work on digital clock synchronisation in the *synchronous* model [28, 25, 26]. One of its main motivations was to gain a better understanding of the linear time/communication complexity barrier that research on pulse synchronisation ran into, without being distracted by the additional timing uncertainties due to communication delay and clock drift. The challenge here is to port these newly developed tools from the synchronous model to the bounded-delay bounded-drift model in a way that keeps them in working condition.

The key to efficiency is a recursive approach, where each node participates in only $\lceil \log f \rceil$ consensus instances, one for each level of recursion. On each level, the overhead of the reduction over a call to the consensus routine is a constant multiplicative factor both in time and bit complexity; concretely, this means that both complexities increase by overall factors of $O(\log f)$. Applying suitable consensus routines yields *exponential improvements* in bit complexity of deterministic and time complexity of randomised solutions, respectively:

1. In the deterministic setting, we exponentially reduce the number of bits each node broadcasts per time unit to $\Theta(\log f)$, while retaining $\Theta(f)$ stabilisation time. This is achieved by employing the phase king algorithm [4] in our construction.
2. In the randomised setting, we exponentially reduce the stabilisation time to $\operatorname{polylog} f$, where each node broadcasts $\operatorname{polylog} f$ bits per time unit. This is achieved using the algorithm by King and Saia [21]. We note that this slightly reduces resilience to $f < n/(3+\varepsilon)$ for any fixed constant $\varepsilon > 0$ and requires private communication channels.
3. In the randomised setting, we can also obtain a stabilisation time of $O(\log f)$, polynomial communication complexity, and optimal resilience of $f < n/3$ by assuming private communication channels. This is achieved using the consensus routine of Feldman and Micali [18]. This almost settles the open question by Ben-Or et al. [3] whether pulse synchronisation can be solved in expected constant time.

The running times of the randomised algorithms (2) and (3) hold with high probability and the additional assumptions on resilience and private communication channels are inherited from the employed consensus routines. Here, private communication channels mean that Byzantine nodes must make their decision on which messages to sent in round r based on knowledge of the algorithm, inputs, and all messages faulty nodes receive up to and including round r. The probability distribution is then over the independent internal randomness of the correct nodes (which the adversary can only observe indirectly) and any possible randomness of the adversary. Our framework does not impose these additional assumptions: stabilisation is guaranteed for $f < n/3$ on each recursive level of our framework as soon as the underlying consensus routine succeeds (within prespecified time bounds) constantly many times in a row. Our results and prior work are summarised in Table 1.

Regardless of the employed consensus routine, we achieve a skew of $2d$, where d is the maximum message delay. This is optimal in our model, but overly pessimistic if the sum of communication and computation delay is not between 0 and d, but from (d^-, d^+), where $d^+ - d^- \ll d^+$. In terms of d^+ and d^-, a skew of $\Theta(d^+ - d^-)$ is asymptotically optimal [29, 34]. We remark that in [20], it is shown how to combine the algorithms from [9] and [34] to achieve this bound without affecting the other properties shown in [9]; we are confident that the same technique can be applied to the algorithm proposed in this work. Finally, all our algorithms work with any clock drift parameter $1 < \vartheta \leq 1.007$, that is, the nodes' clocks can have up to 0.7% drift. In comparison, cheap quartz oscillators achieve $\vartheta \approx 1 + 10^{-5}$.

We consider our results of interest beyond the immediate improvements in complexity of the best known algorithms for pulse synchronisation. Since our framework may employ any consensus algorithm, it proves that pulse synchronisation is, essentially, *as easy* as synchronous consensus – a problem without the requirement for self-stabilisation or any timing uncertainty. Apart from the possibility for future improvements in consensus algorithms carrying over, this accentuates the following open question:

Is pulse synchronisation *at least as hard* as synchronous consensus?

Due to the various lower bounds and impossibility results on consensus [30, 19, 8, 14] mentioned earlier, a positive answer would immediately imply that the presented techniques are near-optimal. However, one may speculate that pulse synchronisation may rather have

the character of (synchronous) approximate agreement [13, 17], as *precise* synchronisation of the pulse events at different nodes is not required. Considering that approximate agreement can be deterministically solved in $O(\log n)$ rounds, a negative answer is a clear possibility as well. Given that all currently known solutions either explicitly solve consensus, leverage techniques that are likely to be strong enough to solve consensus, or are very slow, this would suggest that new algorithmic techniques and insights into the problem are necessary.

2 Preliminaries

Let V denote the set of all n nodes, $F \subseteq V$ be the set of faulty nodes such that $|F| < n/3$, and $G = V \setminus F$ the set of correct nodes. The sets G and F are unknown to the correct nodes in the system. We assume a continous reference time $[0, \infty)$ that is *not* available to the nodes in the distributed system. The reference time is only used to reason about the behaviour of the system. The adversary can choose the initial state of the system (memory contents, initial clock values, any messages in transit), the set F of faulty nodes which it controls, how the correct nodes' clocks progress and what is the delay of each individual message within the respective maximum clock drift and message delay bounds of ϑ and d. We assume that ϑ and d are known constants. For the full formal description of the model we refer to [27].

Pulse synchronisation algorithms. In the pulse synchronisation problem, the task is to have all the correct nodes locally generate pulse events in an almost synchronised fashion, despite arbitrary initial states and the presence of Byzantine faulty nodes. In addition, these pulses have to be well-separated. Let $p(v,t) \in \{0,1\}$ indicate whether a correct node $v \in G$ generates a pulse at time t. Moreover, let $p_k(v,t) \in [t, \infty)$ denote the time when node v generates the kth pulse event at or after time t and $p_k(v,t) = \infty$ if no such time exists. We say that the system has stabilised from time t onwards if

1. $p_1(v,t) \leq t + \Phi^+$ for all $v \in G$,
2. $|p_k(v,t) - p_k(u,t)| < \sigma$ for all $u, v \in G$ and $k \geq 1$,
3. $\Phi^- \leq p_{k+1}(v,t) - \min\{p_k(u,t) : u \in G\} \leq \Phi^+$ for all $v \in G$ and $k \geq 1$,

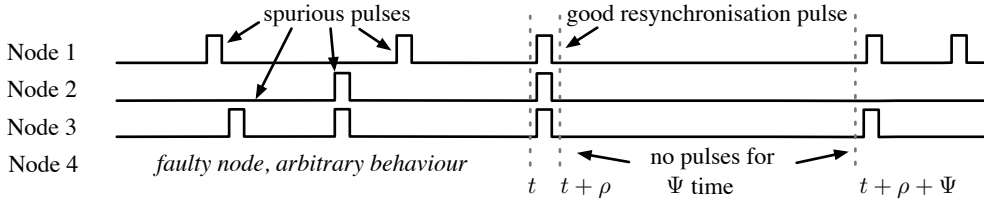
where Φ^- and Φ^+ are the accuracy bounds controlling the separation of the generated pulses. That is, (1) all correct nodes generate a pulse during the interval $[t, t + \Phi^+]$, (2) the kth pulse of any two correct nodes is less than σ time apart, and (3) for any pair of correct nodes their subsequent pulses are at least Φ^- but at most Φ^+ time apart.

We say that $\mathbf{A}$ is an f-resilient pulse synchronisation algorithm with *skew* σ and *accuracy* $\Phi = (\Phi^-, \Phi^+)$ with stabilisation time $T(\mathbf{A})$, if for any choices of the adversary such that $|F| \leq f$, there exists a time $t \leq T(\mathbf{A})$ such that the system stabilises from time t onwards. Moreover, a pulse synchronisation algorithm $\mathbf{A}$ is said to be a *T-pulser* if the accuracy bounds satisfy $\Phi^-, \Phi^+ \in \Theta(T)$. We use $M(\mathbf{A})$ to denote the maximum number of bits a correct node communicates per unit time when executing $\mathbf{A}$.

Resynchronisation algorithms. In our pulse synchronisation algorithm, we use so-called resynchronisation pulses to facilitate stabilisation. Essentially, the resynchronisation pulses are given by a weak variant of a pulse synchronisation algorithm, where the guarantee is that at some point all correct nodes generate a pulse almost synchronously, which is followed by a long period of silence. At all other times, the behaviour can be arbitrary.

Formally, we say that $\mathbf{B}$ is an f-resilient resynchronisation algorithm with skew ρ and separation window Ψ that stabilises in time $T(\mathbf{B})$ if the following holds: for any choices of the adversary such that $|F| \leq f$, there exists a time $t \leq T(\mathbf{B})$ such that every correct

node $v \in G$ locally generates a *resynchronisation pulse* at time $r(v) \in [t, t+\rho)$ and no other resynchronisation pulse before time $t+\rho+\Psi$. We call such a resynchronisation pulse *good*. In particular, we do not impose any restrictions on what the nodes do outside the interval $[t, t+\rho+\Psi)$, that is, there may be *spurious* resynchronisation pulses outside this interval:



3 The transformation framework

Our main contribution is a modular framework that allows us to turn any *non-self-stabilising* synchronous consensus algorithm into a self-stabilising pulse synchronisation algorithm in the bounded-delay model. In particular, this construction yields only a small overhead in time and communication complexity. This shows that efficient synchronous consensus algorithms imply efficient pulse synchronisation algorithms. As our construction is relatively involved, we opt to present it in a top-down fashion.

The main result. For notational convenience, we say that $\mathcal{C}$ is a *family of synchronous consensus routines* with running time $R(f)$ and message size $M(f)$, if for any $f \geq 0$ and $n \geq n(f)$, there exists a synchronous consensus algorithm $\mathbf{C} \in \mathcal{C}$ that runs correctly on n nodes given that there are at most f faulty nodes, terminates in $R(f)$ rounds, and uses messages of size $M(f)$. Here $n(f)$ gives the minimum number of nodes needed as a function of the resilience parameter f. Note that $R(f)$, $M(f)$, and $n(f)$ depend on $\mathcal{C}$; however, making this explicit would clutter notation. We emphasise that the algorithms in $\mathcal{C}$ are not assumed to be self-stabilising. Our main technical result states that given a family of consensus routines, we can obtain pulse synchronisation algorithms with only small additional overhead.

▶ **Theorem 1.** *Let $\mathcal{C}$ be a family of synchronous consensus routines that satisfy (i) for any $f_0, f_1 \in \mathbb{N}$, $n(f_0+f_1) \leq n(f_0)+n(f_1)$ and (ii) both $M(f)$ and $R(f)$ are increasing. Then, for any $f \geq 0$, $n \geq n(f)$, and $1 < \vartheta \leq 1.007$, there exists a $T_0(f) \in \Theta(R(f))$, such that for any $T \geq T_0(f)$ we can construct a T-pulser $\mathbf{A}$ with skew $2d$. The stabilisation time $T(\mathbf{A})$ and number of bits $M(\mathbf{A})$ broadcasted per time unit satisfy*

$$T(\mathbf{A}) \in O\left(d + \sum_{k=0}^{\lceil \log f \rceil} R(2^k)\right) \quad \text{and} \quad M(\mathbf{A}) \in O\left(1 + \sum_{k=0}^{\lceil \log f \rceil} M(2^k)\right),$$

where the sums are empty when $f = 0$.

In the deterministic case, the *phase king algorithm* [5] provides a family of synchronous consensus routines that satisfy the requirements. Moreover, it achieves optimal resilience (i.e., the minimal possible $n(f) = 3f+1$ [30]), constant message size, and asymptotically optimal [19] running time $R(f) \in O(f)$. Thus, this immediately yields the following result.

▶ **Corollary 2.** *For any $f \geq 0$ and $n > 3f$, there exists a deterministic f-resilient pulse synchronisation algorithm over n nodes with skew $2d$ and accuracy bounds $\Phi^-, \Phi^+ \in \Theta(f)$ that stabilises in $O(f)$ time and has correct nodes broadcast $O(\log f)$ bits per time unit.*

Randomised algorithms. Extending Theorem 1 for use with randomised consensus routines is straightforward; the reader is referred to the full paper [27] for details. By applying our construction to a fast and communication-efficient randomised consensus algorithm, e.g. the one by King and Saia [21], we get an efficient randomised pulse synchronisation algorithm.

▶ **Corollary 3.** *Suppose we have private channels. For any $f \geq 0$, constant $\varepsilon > 0$, and $n > (3+\varepsilon)f$, there exists a randomised f-resilient $\Theta(\text{polylog}\, f)$-pulser over n nodes that stabilises in* $\text{polylog}\, f$ *time w.h.p. and has nodes broadcast* $\text{polylog}\, f$ *bits per time unit.*

We can also utilise the constant expected time protocol by Feldman and Micali [18]. With some care, we can show that for $R(f) \in O(1)$, Chernoff's bound readily implies that the stabilisation time is not only in $O(\log n)$ in expectation, but also with high probability.

▶ **Corollary 4.** *Suppose we have private channels. For any $f \geq 0$ and $n > 3f$, there exists a randomised f-resilient $\Theta(\log f)$-pulser over n nodes that stabilises in $O(\log f)$ time w.h.p. and has nodes broadcast* $\text{poly}\, f$ *bits per time unit.*

Proof sketch for Theorem 1. The proof of the main result takes an inductive approach. In the inductive step, we assume two pulse synchronisation algorithms with small resilience. We then use these to construct (via some hoops we discuss later) a new pulse synchronisation algorithm with higher resilience. This step is formalised in the following lemma.

▶ **Lemma 5.** *Let $f, n_0, n_1 \in \mathbb{N}$, $n = n_0 + n_1$, $f_0 = \lfloor (f-1)/2 \rfloor$, and $f_1 = \lceil (f-1)/2 \rceil$. Suppose for $i \in \{0,1\}$ there exists an f_i-resilient $\Theta(R)$-pulser $\mathbf{A}_i$ that runs on n_i nodes and whose accuracy bounds Φ_h^- and Φ_h^+ satisfy $\Phi_h^+ = \varphi\Phi_h^-$ for sufficiently small constants $\varphi > \vartheta$. Let $\mathbf{C}$ be an f-resilient consensus algorithm for a network of n nodes that has running time R and uses messages of at most M bits. Then there exists a $\Theta(R)$-pulser $\mathbf{A}$ that*

- *runs on n nodes and has resilience f,*
- *stabilises in time $T(\mathbf{A}) \in \max\{T(\mathbf{A}_0), T(\mathbf{A}_1)\} + O(R)$,*
- *has nodes broadcast $M(\mathbf{A}) \in \max\{M(\mathbf{A}_0), M(\mathbf{A}_1)\} + O(M)$ bits per time unit, and*
- *has skew $2d$ and whose accuracy bounds Φ^- and Φ^+ satisfy that $\Phi^+ = \varphi\Phi^-$.*

Given the above lemma, it is relatively straightforward to show Theorem 1. Essentially, we can prove the claim for $f \in \bigcup_{k\geq 0}[2^k, 2^{k+1})$ using induction on k. As the base case, we use $f = 0$, that is, pulse synchronisation algorithms that tolerate *no* faulty nodes. These are trivial to obtain for any n: we pick a single node as a leader that generates a pulse when $\Phi^+ - \vartheta d$ time has passed on its local clock. Whenever the leader node pulses, all other nodes observe this within d time units. We have all other nodes generate a pulse whenever they observe the leader node generating a pulse. Thus, for $f = 0$ we have algorithms that stabilise in $O(d)$ time, broadcast $O(1)$ bits in $O(d)$ time, and have accuracy bounds such that $\Phi^- = \Phi^+/\vartheta - d$. For the inductive step, we can assume that f'-resilient pulse synchronisation algorithms exist for all $f' < 2^k$ and $n' \geq n(f')$ and apply Lemma 5.

The auxiliary results. In order to show Lemma 5, we use two main ingredients: (1) a pulse synchronisation algorithm whose stabilisation mechanism is triggered by a resynchronisation pulse and (2) a resynchronisation algorithm providing the latter. These ingredients are formalised in the following two theorems.

▶ **Theorem 6.** *Let $f \geq 0$, $n > 3f$ and $(1+\sqrt{5})/3 > \vartheta > 1$. Suppose for a network of n nodes there exist*

- *an f-resilient synchronous consensus algorithm $\mathbf{C}$, and*
- *an f-resilient resynchronisation algorithm $\mathbf{B}$ with skew $\rho \in O(d)$ and sufficiently large separation window $\Psi \in O(R)$ that tolerates clock drift of ϑ,*

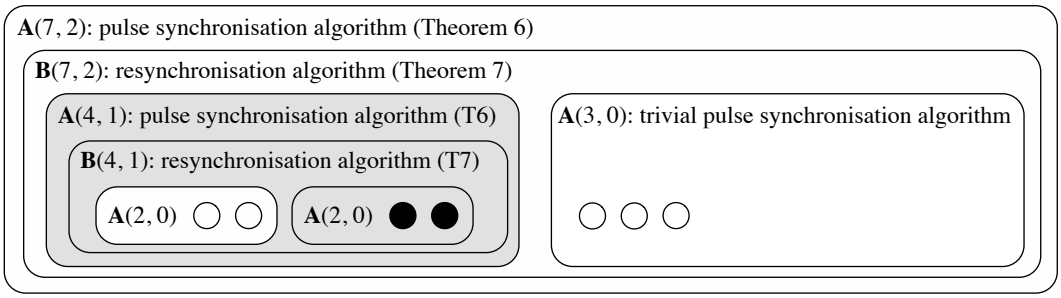


Figure 1 Recursively building a 2-resilient pulse synchronisation algorithm $\mathbf{A}(7,2)$ over 7 nodes. The construction utilises low resilience pulse synchronisation algorithms to build high resilience resynchronisation algorithms which can then be used to obtain highly resilient pulse synchronisation algorithms. Here, the base case consists of trivial 0-resilient pulse synchronisation algorithms $\mathbf{A}(2,0)$ and $\mathbf{A}(3,0)$ over 2 and 3 nodes, respectively. Two copies of $\mathbf{A}(2,0)$ are used to build a 1-resilient resynchronisation algorithm $\mathbf{B}(4,1)$ over 4 nodes using Theorem 7. The resynchronisation algorithm $\mathbf{B}(4,1)$ is used to obtain a pulse synchronisation algorithm $\mathbf{A}(4,1)$ via Theorem 6. Now, the 1-resilient pulse synchronisation algorithm $\mathbf{A}(4,1)$ over 4 nodes is used together with the trivial 0-resilient algorithm $\mathbf{A}(3,0)$ to obtain a 2-resilient resynchronisation algorithm $\mathbf{B}(7,2)$ for 7 nodes and the resulting pulse synchronisation algorithm $\mathbf{A}(7,2)$. 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.

where $\mathbf{C}$ *runs in* $R = R(f)$ *rounds and lets nodes send at most* $M = M(f)$ *bits per round. Then a* $\varphi_0(\vartheta) \in 1 + O(\vartheta - 1)$ *exists so that for any constant* $\varphi > \varphi_0(\vartheta)$ *and sufficiently large* $T \in O(R)$*, there exists an* f*-resilient pulse synchronisation algorithm* $\mathbf{A}$ *for* n *nodes that*

- *has skew* $\sigma = 2d$ *and satisfies the accuracy bounds* $\Phi^- = T$ *and* $\Phi^+ = T\varphi$*,*
- *stabilises in* $T(\mathbf{B}) + O(R)$ *time and has nodes broadcast* $M(\mathbf{B}) + O(M)$ *bits per time unit.*

To apply the above theorem, we require suitable consensus and resynchronisation algorithms. We rely on consensus algorithms from prior work and construct efficient resynchronisation algorithms ourselves. The idea is to combine pulse synchronisation algorithms that have *low resilience* to obtain resynchronisation algorithms with *high resilience*.

▶ **Theorem 7.** *Let* $f, n_0, n_1 \in \mathbb{N}$*,* $n = n_0 + n_1$*,* $f_0 = \lfloor (f-1)/2 \rfloor$*,* $f_1 = \lceil (f-1)/2 \rceil$*, and* $1 < \vartheta \leq 1.007$*. Suppose that for some given* $\Psi \in \Omega(1)$*, sufficiently small constant* $\varphi > \varphi_0(\vartheta)$*, and* $T_0 \in \Theta(\Psi)$*, it holds that for any* $h \in \{0,1\}$ *and* $T_0 \leq T \in O(\Psi)$ *there exists a pulse synchronisation algorithm* $\mathbf{A}_h$ *that*

- *runs on* n_h *nodes and tolerates* f_h *faulty nodes,*
- *has skew* $\sigma = 2d$ *and accuracy bounds* $\Phi_h^- = T$ *and* $\Phi_h^+ = T\varphi$*.*

Then there exists a resynchronisation algorithm $\mathbf{B}$ *with skew* $\rho \in O(d)$ *and separation window of length* Ψ *that generates a resynchronisation pulse by time* $\max\{T(\mathbf{A}_0), T(\mathbf{A}_1)\} + O(\Psi)$*, where nodes broadcast only* $O(1)$ *additional bits per time unit.*

Given a suitable consensus algorithm, one can readily combine Theorems 6 and 7 to obtain Lemma 5. Therefore, we can reduce the problem of constructing an f-resilient pulse synchronisation algorithm to finding algorithms that tolerate up to $\lfloor f/2 \rfloor$ faults and recurse; see Figure 1 for an example on how the two types of algorithms are interleaved.

In the remainder of this paper, we overview the main ideas behind the above two theorems. As the proofs are relatively involved due to a large number of technicalities arising from the

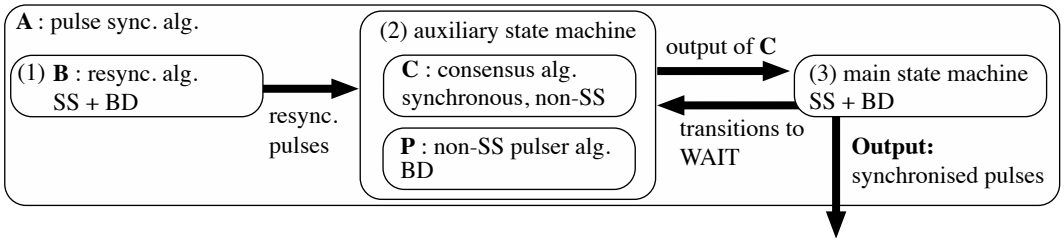


Figure 2 Constructing a self-stabilising (SS) and Byzantine fault-tolerant (BD) pulse synchronisation algorithm **A** out of a Byzantine fault-tolerant but non-stabilising pulse synchronisation algorithm **P**, synchronous consensus algorithm **C**, and resynchronisation algorithm **B**. All algorithms run on the same node set. (1) The resynchronisation algorithm **B** eventually outputs a good resynchronisation pulse, which resets the stabilisation mechanism used by the auxiliary state machine. (2) The auxiliary state machine simulates the executions of **C** using **P**. Simulations are initiated either due to nodes transitioning to a special WAIT state of the main state machine (see Figure 3) or a certain time after a resynchronisation pulse. (3) The main state machine. It generates pulses when a consensus instance outputs "1" and, when stabilised, guarantees re-initialisation of the consensus algorithm by the auxiliary state machine.

uncertainties introduced by the clock drift and message delay, we focus on summarising the key ideas and deliberately skip over a number of details and avoid formalising the claims. All the missing details and full proofs are given in the full paper [27].

4 The self-stabilising pulse synchronisation algorithm (Theorem 6)

We now overview the key elements in the construction of Theorem 6 illustrated in Figure 2:

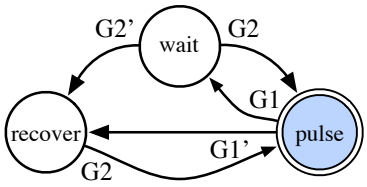
- a non-self-stabilising pulse synchronisation algorithm **P**,
- a synchronous, non-self-stabilising consensus routine **C**,
- a self-stabilising resynchronisation algorithm **B**, and
- the constructed pulse synchronisation algorithm **A**.

Non-self-stabilising pulse synchronisation. The first component we need is a non-self-stabilising pulse synchronisation algorithm **P** that tolerates Byzantine faults. To this end, we use a variant of the classic clock synchronisation algorithm by Srikanth and Toeug [33] that avoids transmitting clock values in favour of unlabelled pulses. As we do not require self-stabilisation for now, we can assume that all nodes receive an *initialisation signal* during the time window $[0, \tau)$ for a given parameter τ. The following theorem summarises the properties of the algorithm.

▶ **Theorem 8.** *Let $n > 1$, $f < n/3$, and $\tau > 0$. If every correct node receives an initialisation signal during $[0, \tau)$, then there exists a pulse synchronisation algorithm* **P** *such that:*

- *all correct nodes generate the first pulse (after initialisation) within time $O(\vartheta^2 d\tau)$,*
- *the pulses have skew $2d$,*
- *the accuracy bounds are $\Phi^- \in \Omega(\vartheta d)$ and $\Phi^+ \in O(\vartheta^2 d)$, and*
- *the algorithm communicates at most one bit per time unit.*

We can simulate synchronous message-passing algorithms with the above algorithm as follows. Assuming that no transient failures or new initialisation signals occur after time τ, by time $O(\vartheta^2 d\tau)$ the algorithm starts to generate pulses with skew $2d$ and accuracy bounds $\Phi^- \in \Omega(\vartheta d)$ and $\Phi^+ \in O(\vartheta^2 d)$. We can set the $\Omega(\vartheta d)$ term to be large enough so that all



Transition guards:

G1 ≡ T_1 expires and saw $\geq n - f$ 'pulse' messages within time T_1 at some point before the timeout expires

G1'≡ T_1 expires and G1 is not satisfied

G2 ≡ auxiliary machine signals 'output 1'

G2'≡ T_{wait} expires or auxiliary machine signals 'output 0'

Figure 3 The main state machine. When a node transitions to state PULSE, it generates a pulse event and sends 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. When a correct node transitions to PULSE, it resets a local timer of length T_1. Once it expires, either Guard G1 or Guard G1' become satisfied. Similarly, the timer T_{wait} is reset when a node transitions to WAIT. Once it expires, Guard G2' is satisfied and the node transitions from WAIT to RECOVER. The node transitions to state PULSE when Guard G2 is satisfied, which requires an "output 1" signal from the auxiliary state machine.

correct nodes can complete local computations and send/receive messages for each simulated round $i - 1$ before the ith pulse occurs. Thus, nodes can associate each message with a distinct round i (by counting locally) and simulate synchronous message-passing algorithms.

The self-stabilising algorithm. The general idea is to repeatedly simulate $\mathbf{C}$ to agree on the time of the next pulse. However, we must deal with an arbitrary initial system state. In particular, the correct nodes may be scattered over the states, with inconsistent memory content, and also the timers employed in the transition guards may have arbitrary values (within their domains). Nonetheless, assume for the moment that there is a small window of length $\rho \in O(d)$ during which each node receives a *resynchronisation pulse*, which triggers the initialisation of the stabilisation mechanism.

The construction relies on two components: (1) a main state machine given in Figure 3 and (2) an auxiliary state machine that acts as a wrapper for an arbitrary consensus algorithm. The main state machine is responsible for generating pulses, whereas the auxiliary state machine generates signals that drive the main state machine. The main machine works as follows: whenever a node enters the PULSE state, it waits for some time to see if at least $n - f$ nodes generated a pulse within a short time window. If not, the system has not stabilised, and the node goes into the RECOVER state to indicate this. Otherwise, the node goes into the WAIT state, where it remains for long enough to (a) separate any subsequent pulses from previous ones and (b) receive the next signal from the auxiliary machine. Once stabilised, the auxiliary machine is guaranteed to send the signal "1" within bounded time. This indicates that the node should pulse again. If no signal arrives on time or the signal is "0", this means that the system has not stabilised and the node goes into the RECOVER state.

While the auxiliary state machine is slightly more involved, the basic idea is simple: (a) nodes try to check whether at least $n - f$ nodes transition to the WAIT state in the main state machine *in a short enough time window* (that is, a time window that would suffice during correct operation) and (b) then use a consensus routine to agree on this observation. Assuming that all correct nodes participate in the simulation of the consensus routine, we get the following:

- If the consensus algorithm $\mathbf{C}$ outputs "0", then some $v \in G$ did not see $n - f$ nodes transitioning to WAIT in a short time window, and hence, the system has not yet stabilised.
- If the consensus algorithm $\mathbf{C}$ outputs "1", then every $v \in G$ agrees that a transition to WAIT happened recently.

In particular, the idea is that when the system operates correctly, the consensus simulation will always succeed and output "1" at every correct node.

The obvious problem here is that the consensus routine is not self-stabilising and it operates in a synchronous model of computation. To remedy the latter problem, we use the algorithm from Theorem 8 to simulate round-based execution. However, this requires that an initialisation signal is generated within a time window of length τ, thus requiring some level of synchrony among the correct nodes. To wiggle our way out of this issue, we carefully construct the main state machine and auxiliary machine to satisfy the following properties:

1. The main state machine guarantees that if *some* correct node transitions to WAIT, then after a short interval no correct node transitions to WAIT for an extended period of time.
2. If a node $u \in G$ sees at least $n - f$ nodes transitioning to WAIT in a short time window (including itself), then the node attempts to start a consensus instance with input "1".
3. If node $u \in G$ attempts to start a simulation of consensus with input "1", then at least $n - 2f > f$ correct nodes $v \in G$ must have recently transitioned to WAIT. As all nodes can reliably detect this event, this essentially ensures that their auxilliary machines synchronise. This way, we can guarantee that all correct nodes initialise a new consensus instance within τ time of each other and generate a consistent output.
4. If this output is "1", all correct nodes generate a synchronised pulse and the system stabilises. Otherwise, all of them transition to state RECOVER.
5. If no $u \in G$ attempts to start a simulation of consensus with input "1" within a certain time, we make sure that all correct nodes end up in RECOVER. Here, we exploit that any consensus instance can be made *silent* [26], which means that no messages are sent by correct nodes if they all have input "0". Hence, even if not all correct nodes actually participate in an instance, it does not matter as long as no correct node has input "1".

Thus, either the system stabilises within a certain time or all correct nodes end up in state RECOVER. This is where we utilise the resynchronisation signals: when a resynchronisation signal is received, the nodes reset a local timer. Since the resynchronisation signal has a small skew of $\rho \in O(d)$, these timers expire within a relatively small time window as well. If the timer expires when all correct nodes are in the RECOVER state, then they can explicitly restart the system in synchrony, also resulting in stabilisation. The key here is to get a good resynchronisation pulse at some point, so that no spurious resynchronisation pulses interfere with the described stabilisation mechanism until it is complete. Once succesful, no correct nodes transition to RECOVER anymore. Thus, any subsequent resynchronisation pulses do not affect pulse generation. For a detailed discussion and formal analysis, see [27].

5 Generating resynchronisation pulses (Theorem 7)

The final ingredient is a mechanism to generate resynchronisation pulses; see Figure 4 for the general structure of the construction. Recall that a *good* resynchronisation pulse is an event triggered at all correct nodes within a small time interval, followed by at least Ψ time during which no correct node triggers a new such event. In order to construct an algorithm that generates such an event, we partition the set of n nodes into two disjoint *blocks* of roughly $n/2$ nodes. Each block runs an instance of a pulse synchronisation algorithm tolerating f_i faults, where $f_0 + f_1 + 1 = f$ (and $f_0 \approx f_1 \approx f/2$). For these two algorithms, we choose different pulsing frequencies (that is, accuracy bounds) that are roughly coprime integer multiples of the desired separation window Ψ. Both algorithms are used as potential sources of resynchronisation pulses. The idea behind our construction is illustrated in Figure 5. If both instances stabilise, it is not difficult to set up the frequencies such that $\mathbf{A}_i$ eventually generates a pulse that is not followed by a pulse from $\mathbf{A}_{1-i}$ within time Ψ.

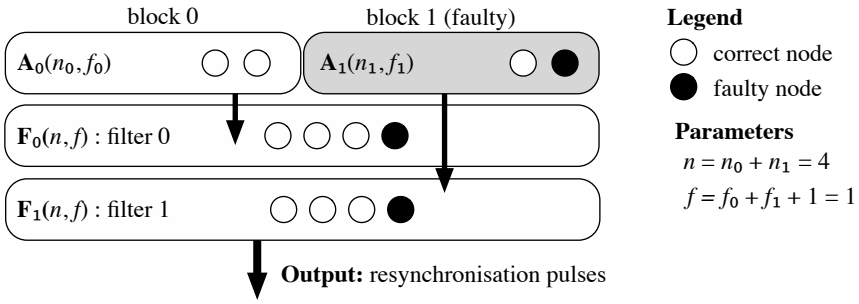


Figure 4 Construction of an f-resilient resynchronisation algorithm on n nodes from f_i-resilient pulse synchronisation algorithms on n_i nodes, where $f = f_0 + f_1 + 1$ and $n = n_0 + n_1$. The n nodes are divided into two groups of n_0 and n_1 nodes. These groups run pulse synchronisation algorithms $\mathbf{A}_0$ and $\mathbf{A}_1$, respectively. At least one of these algorithms is guaranteed to stabilise eventually. Here, $\mathbf{A}_1$ (gray block) has too many faulty nodes and does not stabilise. All of the n nodes together run two filtering mechanisms $\mathbf{F}_0$ and $\mathbf{F}_1$ for the outputs of $\mathbf{A}_0$ and $\mathbf{A}_1$, respectively. These ensure that no correct node locally generates a resynchronisation pulse without all correct nodes registering this event, and then apply timeout constraints to enforce the desired frequency bounds.

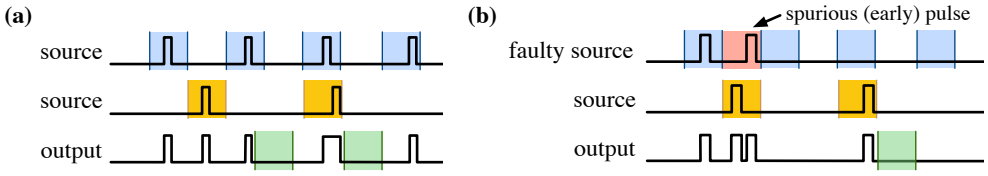


Figure 5 Idea of the resynchronisation algorithm. We take two pulse sources with (up to scaling) coprime frequencies and output the logical OR of the two sources. In this example, the pulses of the first source should occur in the blue regions, whereas the pulses of the second source should hit the yellow regions. The green regions indicate a period where a pulse from either source is followed by at least Ψ time of silence. Eventually, such a region appears. (a) Two correct sources that pulse with set frequencies. (b) One faulty source that produces spurious pulses. Here, a pulse occurs too early (red region), and thus, we then enforce that the faulty source is silenced for $\Theta(\Psi)$ time.

Unfortunately, one of the instances (but not both) could have more than f_i faulty nodes, never stabilise, and thus generate possibly inconsistent pulses at arbitrary points in time. We overcome this by a two-step filtering process illustrated in Figure 6. First, we apply a number of threshold votes ensuring that if a pulse of a block is considered as a candidate resynchronisation pulse by *some* correct node, then *all* correct nodes observe this event. Second, we locally *filter out* any observed events that do not obey the prescribed frequency bounds for the respective block. Thus, the faulty block either generates (possibly inconsistent) pulses within the prescribed frequency bounds only, or its influence is suppressed entirely (for sufficiently long time). Either way, the correctly operating block will eventually succeed in generating a resynchronisation pulse. Further details and all missing proofs appear in the full version of this paper [27].

Acknowledgements. We are grateful to Danny Dolev for numerous discussions on the pulse synchronisation problem and detailed comments on early drafts of this paper. We also wish to thank Borzoo Bonakdarpour, Janne H. Korhonen, Christian Scheideler, Jukka Suomela, and anonymous reviewers for their helpful comments.

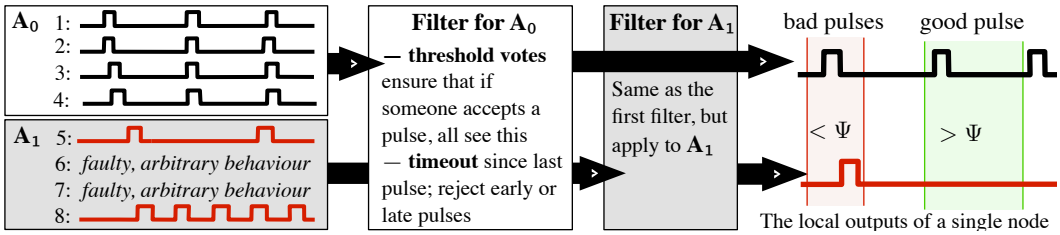


Figure 6 Example of the resynchronisation construction for 8 nodes tolerating 2 faults. We partition the network into two parts, each running a pulse synchronisation algorithm $\mathbf{A}_i$. The output of $\mathbf{A}_i$ is fed into the respective filter and any pulse that passes the filtering is used as a resynchronisation pulse. The filtering consists of (1) having *all* nodes in the network participate in a threshold vote to see if anyone thinks a pulse from $\mathbf{A}_i$ occurred (i.e. enough nodes running $\mathbf{A}_i$ generated a pulse) and (2) keeping track when was the last time a pulse from $\mathbf{A}_i$ occurred to check that the accuracy bounds of $\mathbf{A}_i$ are respected: pulses that appear too early or too late are ignored.

References

1 M. K. Aguilera and S. Toueg. Simple bivalency proof that t-resilient consensus requires $t+1$ rounds. *Information Processing Letters*, 71(3):155–158, 1999.

2 M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *PODC 1983*, pages 27–30, 1983.

3 M. Ben-Or, D. Dolev, and E. N. Hoch. Fast self-stabilizing Byzantine tolerant digital clock synchronization. In *PODC 2008*, pages 385–394, 2008.

4 P. Berman, J. A. Garay, and K. J. Perry. Towards optimal distributed consensus. In *FOCS 1989*, pages 410–415, 1989.

5 P. Berman, J. A. Garay, and K. J. Perry. Bit optimal distributed consensus. In *Computer Science: Research and Applications*, pages 313–321, 1992.

6 A. Daliot, D. Dolev, and H. Parnas. Self-stabilizing pulse synchronization inspired by biological pacemaker networks. In *SSS 2003*, pages 32–48, 2003.

7 Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

8 D. Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982.

9 D. Dolev, M. Függer, C. Lenzen, and U. Schmid. Fault-tolerant algorithms for tick-generation in asynchronous logic. *Journal of the ACM*, 61(5):30:1–30:74, 2014.

10 D. Dolev, J. Y. Halpern, and H. R. Strong. On the possibility and impossibility of achieving clock synchronization. *Journal of Computer and System Sciences*, 32(2):230–250, 1986.

11 D. Dolev, K. Heljanko, M. Järvisalo, J. H. Korhonen, C. Lenzen, J. Rybicki, J. Suomela, and S. Wieringa. Synchronous counting and computational algorithm design. *Journal of Computer and System Sciences*, 82(2):310–332, 2016.

12 D. Dolev and E. N Hoch. Byzantine self-stabilizing pulse in a bounded-delay model. In *SSS 2007*, pages 234–252, 2007.

13 D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, 1986.

14 D. Dolev and R. Reischuk. Bounds on information exchange for Byzantine agreement. *Journal of the ACM*, 32(1):191–204, 1985.

15 S. Dolev. *Self-Stabilization*. Cambridge, MA, 2000.

16 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.

17 A. D. Fekete. Asymptotically optimal algorithms for approximate agreement. *Distributed Computing*, 4(1):9–29, 1990.

18 P. Feldman and S. Micali. Optimal algorithms for Byzantine agreement. In *STOC 1988*, pages 148–161, 1988.

19 M. J. Fischer and N. A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982.

20 P. Khanchandani and C. Lenzen. Self-stabilizing Byzantine clock synchronization with optimal precision. In *SSS 2016*, pages 213–230, 2016.

21 V. King and J. Saia. Breaking the $O(n^2)$ bit barrier. *Journal of the ACM*, 58(4):1–24, 2011.

22 L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(I):52–78, 1985.

23 L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

24 C. Lenzen, M. Függer, M. Hofstätter, and U. Schmid. Efficient construction of global time in SoCs despite arbitrary faults. In *DSD 2013*, pages 142–151, 2013.

25 C. Lenzen and J. Rybicki. Efficient counting with optimal resilience. In *DISC 2015*, pages 16–30, 2015.

26 C. Lenzen and J. Rybicki. Near-optimal self-stabilising counting and firing squads. In *SSS 2016*, pages 263–280, 2016.

27 C. Lenzen and J. Rybicki. Self-stabilising Byzantine clock synchronisation is almost as easy as consensus, 2017. Full version. URL: `http://arxiv.org/abs/1705.06173`.

28 C. Lenzen, J. Rybicki, and J. Suomela. Towards optimal synchronous counting. In *PODC 2015*, pages 441–450, 2015.

29 J. Lundelius and N. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2–3):190–204, 1984.

30 M. C. Pease, R. E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

31 M. O. Rabin. Randomized Byzantine generals. In *FOCS 1983*, pages 403–409, 1983.

32 M. Raynal. *Fault-tolerant agreement in synchronous message-passing systems*. Morgan & Claypool, 2010.

33 T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, 1987.

34 J. L. Welch and N. Lynch. A new fault tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, 1988.