Generation of a fault-tolerant clock through redundant crystal oscillators

Wolfgang Dür a, Matthias Függer b, Andreas Steininger a,∗

a TU Wien, Institute of Computer Engineering, A-1040 Vienna, Austria
b CNRS & LSV, ENS Paris-Saclay, Université Paris-Saclay & Inria, France

ARTICLE INFO
Keywords:
Fault tolerance
Single point of failure
Fault tolerant clocking
Formal proof

ABSTRACT
Having a precise and stable clock that is still fault tolerant is a fundamental prerequisite in safety critical real-time systems. However, combining redundant independent clock sources to form a unified fault-tolerant clock supply is non-trivial, especially when redundant clock outputs are required – e.g., for supplying the replicated nodes within a TMR architecture through a clock network that does not suffer from a single point of failure. Having these outputs fail independent but still keeping them tightly synchronized is highly desirable, as it substantially eases the design of the overall architecture.

In this paper we address exactly this challenge. Our approach extends an existing, ring-oscillator like distributed clock generation scheme by augmenting each of its constituent nodes with a stable clock reference. We introduce the appropriately modified algorithm and illustrate its operation by simulation experiments. These experiments further demonstrate that the four clock outputs of our circuit do not share a single point of failure, have small and bounded skew, remain stabilized to one crystal source during normal operation, do not propagate glitches from one failed clock to a correct one, and only exhibit slightly extended clock cycles during a short stabilization period after a component failure. In addition we give a rigorous formal proof for the correctness of the algorithm on an abstraction level that is close to the implementation.

1. Introduction

Computers are being entrusted with safety-critical functions in a rapidly increasing number of applications, with autonomous vehicles being just one recent example. Consequently fault tolerance is essential for these systems. While a lot of alternative fault-tolerance techniques are available, (coarse-grain) triple-modular redundant (TMR) architectures have gained much popularity. This is partly due to the high error detection coverage they can attain through their “output centric” approach: No matter what the actual cause may be – the voter just takes the majority of matching outputs and masks the faulty one. Another beneficial feature of TMR is its simplicity: The redundant nodes can be off-the-shelf components (or IP modules) without any special features or extensions.

One threat to TMR architectures is the so-called common-mode failure: If two of the three redundant nodes fail in the same way, the voter will decide for the erroneous result. That is why in conservative designs the redundant nodes are often independent PCBs. However, it is very appealing to use TMR on-chip as well, in the shape of replicated IP modules. This not only provides cost savings, but also performance benefits, as the replica have efficient communication with the voter. Ideally the whole architecture is operated in lock-step, which significantly simplifies the voter. However, at this point the clock potentially becomes a single point of failure, unless it can be furnished with fault tolerance as well.

Building such a fault-tolerant clock to supply the replica within a TMR system is challenging, as it involves a fundamental conflict between using independent clock sources on the one hand and attaining the desired synchrony between the replica on the other hand. The solution we present in this paper addresses exactly this challenge.

In Section 2 we will briefly review and discuss existing approaches, before elaborating the requirements we want a solution to meet in Section 3. Subsequently, we will briefly introduce the DARTS approach that our solution builds upon, along with the extensions we propose, in Section 4. Next, Section 5 will present a formal correctness proof for our algorithm. Finally, as a practical proof of concept, and to show the limitations, we will discuss some selected simulation results in Section 6, before we conclude the paper in Section 7.
2. Background and related work

A very straightforward way of making a clock fault tolerant is to augment a primary clock with an error detector and switch over to a redundant source once the primary clock fails [1–9]. This method is, e.g., also used in Intel’s STRATIX10 FPGAs.\(^1\) The challenges with this approach are (a) to avoid glitches upon the switch-over and (b) to mitigate metastability at the domain crossing between supervised clock and reference time of the error detector. While (a) can be handled by special filter circuits [10,11], (b) requires care in the design and selection of the detection method [12].

While these approaches work fine for handling failure of the clock source, the fundamental issue with all of them is that they necessarily suffer from a single point of failure, like the error detector, the switch or the voter. And ultimately the single clock output is another weakness. Therefore schemes with multiple clock outputs become attractive. Simply using redundant clock sources in parallel solves the fault-tolerance issue, but causes problems with the synchronization: If the independent redundant clocks are, e.g., supplied to the replicated nodes in a TMR architecture, the activities of these nodes will become uncorrelated as a consequence of the clock mismatch and drift. So a synchronization of the clocks is definitely desirable.

Obviously, stable independent clock sources like crystal oscillators cannot be directly synchronized, as they do not allow for a (sufficient) adjustment of their phase or frequency. There are clock synchronization algorithms where a certain number \(n\) of microticks of such local oscillator is counted to generate macroticks, and the latter are then globally synchronized through a distributed algorithm that continuously adjusts the local values for \(n\) as appropriate [13]. While this approach works well for coarse-grain synchronization in distributed systems, it cannot provide the fine-grain synchronization required for lock-step operation of IP modules.

For this fine level of granularity, diverse implementations of distributed ring oscillators have been proposed [14–16]. While all these approaches can produce an arbitrary number of mutually synchronized local clocks with a jitter of a few cycles at most, they have the common drawback that the clock frequency is determined by path delays alone and hence neither accurate nor stable. This is disadvantageous for applications requiring a notion of real time or when signal sampling is performed.

In Section 4 of this paper we will build on the DARTS approach [16] that resembles a hardware implementation of a distributed clock synchronization algorithm in asynchronous hardware and hence falls into the class of ring-oscillator based solutions. Following the principle already outlined in [17], we will augment it with stable crystal clock sources and make the whole system follow these references while still maintaining its fault tolerance properties. As an extension of [17], however, we will elaborate the algorithm formally and in great detail, and we will give a rigorous formal correctness proof for it.

3. Requirements

Our envisioned use case is a TMR system whose redundant nodes shall be supplied with a clock that does not constitute a single point of failure. Consequently, the very convenient and popular solution of having the nodes operate in lock-step supplied by a single clock source does not work. Simply using independent clock sources will make the nodes run at different speed and, even if the difference is small, this will cause a significant time offset after a long time of operation. So even if the voter could accommodate this offset, the nodes may see different inputs for the same operation, just because of the increasing time offset, and hence produce non-matching results even in the fault-free case. Compensating that would entail some sort of synchronization among the nodes, which is undesired, since ideally the nodes should be unaware of being part of a TMR architecture. Also, communication (e.g., with the voter) requires buffering, with the necessary buffer size growing over time, essentially towards infinity (unless synchronized). So we want the nodes’ clocks to remain synchronized.

This leads to the following list of requirements:

- (R1) Tolerance against failure of a clock source: any type of misbehavior of a single clock source must not impede the correct operation of more than one clock output.
- (R2) No single point of failure: the failure of any single component in the clocking infrastructure (circuit or interconnect) must not impede the correct operation of more than one clock output.
- (R3) Accuracy: The clock frequency on each node must always remain in the interval spanned by the slowest and the fastest source frequency. This allows to establish a much better accuracy and stability than with ring-oscillator based solutions.
- (R4) Synchrony/precision: The clocks provided to the individual nodes may have a phase offset (skew), but this offset must have an upper bound. With an offset bounded to \(k\) clock cycles, a communication buffer of size \(k\) is sufficient without further provisions for backpressure or synchronization. This stands in sharp contrast to the case of clocks that drift apart, causing essentially unbounded skew.
- (R5) No glitching: for a hardware clock it is an important property to have a half period that is always above a defined minimum value \(H_{\text{min}}\). Shorter half periods (pulses) will be perceived by the driven circuit as glitches that violate timing assumptions (comparable to operating at an excessive clock frequency).

4. Proposed solution for redundant clock outputs

4.1. Starting point: the DARTS approach

The DARTS architecture (Distributed Algorithm for Robust Tick Synchronization) implements the formally proven fault-tolerant tick synchronization algorithm (TSA) from Srikanth and Toueg [18] in asynchronous hardware. More specifically, instances of this algorithm are implemented in \(n\) nodes that will further be called “TS nodes”. According to the TSA these nodes communicate to agree on jointly progressing their local time (represented as a tick counter). With \(n \geq 3f + 1\) this arrangement can tolerate \(f\) arbitrarily failing TS nodes; in distributed computing this fault model is precisely called “Byzantine failure”. The TS nodes need to be fully connected, i.e., there must be an individual communication link from one TS node to each other. Based on its local status, which is constituted by its own tick count and the (local) knowledge about all other TS nodes’ tick counts, each TS node proposes, under certain conditions, for all other TS nodes and itself to increment the tick count. The condition (increment rule) is, roughly speaking, that sufficiently many TS nodes agree on that. In this context “sufficiently many” means that all but the assumed maximum number of faulty TS nodes agree, which is \(n - f\), or, for the case of \(n = 3f + 1\) that we further assume, this equals \(2f + 1\). In addition, a TS node may increment its local tick counter if it sees that at least one non-faulty other TS node, that is \(f + 1\) overall, already arrived at this value (relay rule). For details on the algorithm see the original paper [18]. As a result of executing this algorithm, all TS nodes will continuously increment their local tick count (time) in synchrony with all other, non-faulty ones. This synchrony is expressed by a limit on the maximum skew between the TS nodes, which is determined by the communication delays.

For the hardware implementation of this algorithm several things had to be adapted. In particular the unbounded count that represents the absolute local time had to be removed, and up/down counters were used to represent the local time relative to the respective other TS nodes. For implementing these up/down counters in an asynchronous and

---

metastability-safe way, Muller pipelines [19] were carefully inter-connected. Each of the TS nodes comprises \( n = 3f + 1 \) such pipelines, each representing the relative position of the local count to that on one of the other TS nodes.\(^2\) Observing the fill level of all of these pipelines allows for executing the above two rules to generate an output transition. Instead of sending a count value, like in the original algorithm, the hardware implementation just produces a next clock edge. More specifically, to produce a next rising (falling) edge, a TSA must either have received falling (rising) edges from at least \( 2f + 1 \) TS nodes at its inputs, or already rising (falling) edges from at least \( f + 1 \) TS nodes.\(^3\) Overall this yields a local clock that stays in synchrony with all others. For details on the hardware implementation and the resulting properties we refer to [16].

Note that in DARTS there is no time reference involved – the oscillation underlying the generated clock results plainly from the communication among the TS nodes. Consequently the frequency is determined by propagation delays through gates and interconnect alone, and varies with supply voltage and temperature, as well as being process variation dependent. So in essence, while fulfilling requirements (R1), (R2), (R4) and (R5), DARTS does not fulfill (R3).

4.2. Proposed extension with stable sources

Our idea is to extend the existing DARTS scheme by stable sources, like crystal oscillators, to improve its accuracy and hence meet (R3). Since our envisioned use case is to supply the clock for a TMR system, we make the following assumptions and restrictions beyond those made for DARTS:

- (A1) We reduce the fault model to the case of \( f = 1 \), which yields 4 TS nodes in the DARTS architecture. This is just to simplify the explanation and the simulation. Our algorithm and the respective proofs are still general, for any choice of \( f \).
- (A2) Unlike DARTS, we assume that the TS nodes are not distributed but close together in a compact clock generator circuit block.
- (A3) As a consequence of (A2) the communication paths between the TS nodes are fast, also the TS nodes remain relatively lean and can operate fast.

---

\(^2\) Each TS node also has a loop back to itself, i.e. uses its own output as an input.

\(^3\) More precisely, considering that there are buffers involved and the skew may become larger than one period, these edges must belong to the appropriate wave.
• (A4) As a further consequence of (A2) we also assume symmetric layout with the delay mismatch among the TS nodes and paths being relatively small.
• (A5) We assume the reference sources to produce a relatively stable clock. Abrupt changes in their frequency are considered a failure (and can be tolerated as such).
• (A6) A faulty component can have arbitrary behavior in the digital domain, but non-digital behavior like intermediate "analog" voltage levels that are neither associated to a clean logic HI nor a logic LO are not considered. This assumption had already been made for DARTS, like for most fault-tolerant systems.

The basic idea is to augment each TS node with an own stable local reference oscillator (LRO), typically that will be a crystal oscillator. All these oscillators have the same nominal frequency. Next, we extend the increment rule in such a way that it only fires when the LRO also indicates that a new transition is due. To make this work, we leverage (A3) to argue that the original TSA executes fast enough to already "arm" the increment rule, and the LRO determines the point in time when it fires.

4.3. Formalization of the modified algorithm

More precisely, we implement Algorithm 1 as shown below. Like in DARTS we assume that each TS node i has each of its inputs connected to one of the other TS nodes’ output (plus its own output). A counter c_{i,j} located at each such input maintains TS node i’s view of how many ticks TS node j already generated. Since these c_{i,j} are up/down counters, counting up when a remote tick is received (i.e. from the other TS node) and counting down each time i locally issues a tick, this view is relative, i.e. it gives visibility how many ticks j is ahead of i or lagging.

Similarly, lines 7...13 express how, by the operator counter_{\[\delta_a,\delta_b\]}(i,j), a counter c_{i,j} maintained on TS node i is incremented upon reception of a remote tick from TS node j. Before the actual increment operation the cut-off is implemented (in this case to a maximum of 3). This counter management is formalized in lines 1...32 of Algorithm 4.3. More specifically, lines 1...5 express how, by the operator counter_{\[\delta_a,\delta_b\]}(i,j), a counter c_{i,j} maintained on TS node i is incremented upon reception of a remote tick from TS node j. Before the actual increment operation the cut-off is implemented (in this case to a maximum of 3). Note that the assignment of the incremented value is associated with an "after" attribute, which accounts for the non-zero delay of the physical implementation of the counters (and especially the possibly non-matching delays across the individual elastic pipelines that constitute them).

Algorithm 1

1: define counter_{\[\delta_a,\delta_b\]}(i,j)
2:  if c_{i,j} < 3 then
3:     c_{i,j} ← c_{i,j} + 1 after [\delta_a,\delta_b]
4:  end if
5: end define
6:  if c_{i,j} > -2 then
7:     c_{i,j} ← c_{i,j} - 1 after [\delta_a,\delta_b]
8:     next_pol_{i,j} ← -next_pol_{i,j}
9: else
10:       next_pol_{i,j} ← -next_pol_{i,j} after [\delta_a,\delta_b]
11:  end if
12: end define
13: define counter_{LRO}_{\[\delta_a,\delta_b\]}(i)
14:  if c_{LRO} < 2 then c_{LRO} ← c_{LRO} + 1 after [\delta_a,\delta_b]
15: end if
16: define counter_{LRO}_{\[\delta_a,\delta_b\]}(i)
17:  if c_{LRO} > -2 then
18:     c_{LRO} ← c_{LRO} - 1 after [\delta_a,\delta_b]
19:     next_pol_{LRO} ← -next_pol_{LRO}, after [\delta_a,\delta_b]
20: else
21:       next_pol_{LRO} ← -next_pol_{LRO}, after [\delta_a,\delta_b]
22: end if
23: define LRO_tick(i)
24:    wait for time t ∈ [T_{min},T_{max}]
25:    counter_{LRO}_{\[\delta_a,\delta_b\]}(i)
26: end define
27: variables
28: cur_pol_{i} : bool ← 0 \ (i \in P)
29: next_pol_{i,j} : bool ← 1 \ (i,j \in P)
30: next_pol_{LRO} : bool ← 1 \ (i \in P)
31: c_{i,j} : integer ← 0 \ (i,j \in P)
32: c_{LRO} : integer ← 0 \ (i \in P)
33: end variables
34:  foreach i \in P do repeatedly in parallel
35:    ready_{LRO} ← cur_pol_{i} \neq next_pol_{LRO} \land c_{LRO} > 0
36:    \forall j \in P : rdcrelay_{rem,j} ← cur_pol_{j} \neq next_pol_{j} \land c_{i,j} > 0
37:    \forall j \in P : rdpolprop_{rem,j} ← cur_pol_{i} \neq next_pol_{j} \land c_{i,j} \geq 0
38:    \forall i \in P : counter_{\[\delta_a,\delta_b\]}(i,j)
39:    \forall j \in P : counter_{\[\delta_a,\delta_b\]}(j,i)
40: end if
41:  end foreach
42:  if progress_rule_{i} \lor relay_{rule_{i}} then
43:    cur_pol_{i} ← \neg cur_pol_{i}
44:  \forall j \in P : counter_{\[\delta_a,\delta_b\]}(i,j)
45:    counter_{LRO}_{\[\delta_a,\delta_b\]}(i)
46:    \forall j \in P : counter_{\[\delta_a,\delta_b\]}(j,i)
47: end if
48: LRO_tick(i)
49: end foreach

This, however, is the ideal case that does not yet consider the synchronization with the other TS nodes. More details will become clear later on.

4.3. Formalisation of the modified algorithm

More precisely, we implement Algorithm 1 as shown below. Like in DARTS we assume that each TS node i has each of its inputs connected to one of the other TS nodes’ output (plus its own output). A counter c_{i,j} located at each such input maintains TS node i’s view of how many ticks TS node j already generated. Since these c_{i,j} are up/down counters, counting up when a remote tick is received (i.e. from the other TS node) and counting down each time i locally issues a tick, this view is relative, i.e. it gives visibility how many ticks j is ahead of i or lagging.

Here we also model an overrun and underrun of these counters, as this cut-off behavior corresponds to the implementation by an elastic pipeline that just ignores transitions exceeding its depth.

This counter management is formalized in lines 1...32 of Algorithm 4.3. More specifically, lines 1...5 express how, by the operator counter_{\[\delta_a,\delta_b\]}(i,j), a counter c_{i,j} maintained on TS node i is incremented upon reception of a remote tick from TS node j. Before the actual increment operation the cut-off is implemented (in this case to a maximum of 3). Note that the assignment of the incremented value is associated with an “after” attribute, which accounts for the non-zero delay of the physical implementation of the counters (and especially the possibly non-matching delays across the individual elastic pipelines that constitute them).
indicate, by activating also being active. This very extension allows the local LRO to suspend (see later). In hardware this is implemented by separating the evaluation of the reference oscillator (LRO) associated with each TS node. The op \( c_{\text{LRO}} \) is incremented by each transition of the reference oscillator through the operator \( \text{LRO}_{\text{tick}}(i) \). Note that this is accompanied by delaying the completion of the operation for one half period \( \Pi_0 \) of the LRO clock through the wait statement.

With these operating being defined, the actual algorithm can start with a synchronization (lines 34..40): All counters \( \{c_{i,j} \}_{i,j} \) are initialized to zero, all clock outputs \( \text{cur}_{\text{pol}}(i) \) to logic low, and the handshake flags \( \text{next}_{\text{pol}}(i) \) to logic high.

Next, the counter states are evaluated: In line 43 the statement \( c_{\text{LRO}} > 0 \) checks whether the LRO counter received more transitions from the LRO than have been sent locally – in which case the LRO pipeline would indicate, by activating \( \text{ready}_{\text{LRO}} \), the readiness for firing a new local tick. This, however only happens in case the handshake allows for that: The condition \( \text{cur}_{\text{pol}} \neq \text{next}_{\text{pol}}(i) \) ensures that the current value of the LRO counter is qualified for generating the next clock transition, namely the one leading to the logic level indicated by \( \text{next}_{\text{pol}}(i) \), and has not already been used for generating the previous transition that led to the level indicated by \( \text{cur}_{\text{pol}}(i) \). As already mentioned above, this ensures that a rule can only fire once.

In an analogous way the counters \( c_{i,j} \) are checked for containing at least 0 \( \text{rdy}_{\text{relay}_{\text{rem}}} \) or more than 0 \( \text{rdy}_{\text{progress}_{\text{rem}}} \) remote transitions (more than local ones) in lines 44 and 45, again considering the respective handshake flags.

Based on the condition flags thus generated the actual rules can be evaluated. In line 46 the progress rule is executed, just like in DARTS: If on one TS node \( i \) at least \( 2f+1 \) inputs \( j \) see their \( \text{rdy}_{\text{progress}_{\text{rem}}} \) activated, the progress rule makes that TS node fire. However, the specific feature of our algorithm is that this does not happen without \( \text{ready}_{\text{LRO}} \) also being active. This very extension allows the local LRO to suspend the firing until its own transition occurs, thus achieving the desired synchronization.

In line 47 the relay rule is formulated: It fires when more than \( f+1 \) inputs see their \( \text{rdy}_{\text{relay}_{\text{rem}}} \) activated – this time, however, without considering the LRO. This is because it does not make sense to slow down TS nodes that are “pulled behind” anyway, by synchronizing them with their LRO.

Finally, in line 49 the two rules are combined (ORed) and the appropriate actions performed: The local tick is generated by toggling the output polarity of that TS node \( i \) in line 50; in consequence all local counters of \( i \) are decremented (line 52), as well as the associated LRO counter (line 53); and the transition is forwarded to all other TS nodes where it increments the remote counters (line 55) associated to it.

Notice the delay parameters that are passed to the operators upon call: For the local counters the delay \( \delta \) is in the interval \( [\delta_{\text{min}}, \delta_{\text{max}}] \) and includes a delay element that is deliberately introduced to guarantee a lower limit for the pulse width of the generated clock even under worst circumstances (see later); and for the remote counters this is a delay \( \Delta \) from the interval \( [\Delta_{\text{min}}, \Delta_{\text{max}}] \) that solely is constituted by transmission delays on the interconnect lines.

Further note that we assume all the rules in lines 43..47 to be executed concurrently on a TS node. The if clause in lines 47..54 is assumed to be executed concurrently with those as well, while the statements within the clause are sequentially executed. The handshake established by \( \text{cur}_{\text{pol}} \) and \( \text{next}_{\text{pol}} \) ensures an appropriate interleaving of the concurrent executions.

In the same way the LRO clock expressed by the statement \( \text{LRO}_{\text{tick}} \) (in line 58) is executed concurrently to the others. Here an independent operation is desired (free running clock) rather than an interleaving, and the wait statement in the LRO tick operator restricts the execution of the statement to a single invocation just at each clock edge.

On top of the concurrency on a single TS node, as described above, all TS nodes execute that whole algorithm concurrently.

### 4.4. Steady-state operation of the extended algorithm

In the practical application we have 4 crystal oscillators with clock frequencies \( f_1, f_2, f_3, f_4 \) which are all nominally the same, but of course there is a certain mismatch through tolerances. Without loss of generality, let us assume \( f_1 > f_2 > f_3 > f_4 \). Let us furthermore assume the system is initialized to a state where all TS nodes are already armed, waiting for their LRO to fire. In the first round this may work well, but then over time the fastest LRO, the one running at \( f_1 \), will try to fire its TS node \( (\text{TS}_1) \) even before the TSA has found a majority of other TS nodes which agree on the next clock edge to be fired (let us assume it is a rising edge). In our case of \( f = 1 \) this means that \( \text{TS}_1 \) will have to wait for \( \text{TS}_2 \) and \( \text{TS}_3 \) before it gets released to fire. Even though \( \text{TS}_3 \) receives this rising transition, it still cannot fire: For executing the relay rule that one transition is too little (it needs at least \( 2f \)); for the increment rule to work, it still needs to receive a transition from its LRO. When the transition from \( \text{LRO}_3 \) finally arrives, \( \text{TS}_2 \) will fire. Now \( \text{TS}_2 \) and \( \text{TS}_4 \), who had already received the rising transition from \( \text{TS}_3 \) also receive the one from \( \text{TS}_2 \), which is sufficient to fire without waiting for the LRO. Once the rising transitions thus generated by \( \text{TS}_3 \) and \( \text{TS}_4 \) are received by \( \text{TS}_1 \), the latter can execute its increment rule to arm its TSA for the next falling transition. Since \( \text{LRO}_3 \) had already delivered the required transition (and made \( c_{\text{LRO}} > 0 \)), this firing will happen immediately and the next cycle begins.

This cyclic process has several consequences:

- The speed of \( \text{TS}_2 \) will dictate the overall system speed. The clock outputs all follow \( f_2 \). For an illustration see Fig. 2 in Section 6.
- As a consequence the fastest LRO, the one at \( \text{TS}_1 \), runs ahead and gets out of sync. We therefore need to buffer at least one of its transitions to make \( \text{TS}_1 \) fire once its TSA allows for it. There is no need to make the buffer deeper; it does not matter when it overruns. In fact we will see later that a deeper buffer is even counter-productive in the transient phase after the failure of a source.
- Naturally the slower LROs, those of \( \text{TS}_3, \text{TS}_4 \), also get out of sync. Here the relay rule takes care to enforce a clock transition even before a matching transition of the LRO has been seen. This happens, when at least two \( (f + 1) \) of the faster TS nodes have fired a matching transition already. Here it is important to cancel each arriving matching transition from the local clock source, as it is too late and hence would unduly cause a firing in a next round. This is accomplished though buffering at least one input transition, such that an incoming transition from the LRO is removed from the pipeline (as a function of the difference counter), as soon as it arrives. Again an overflow does not matter here.

Should one of the clock sources or the TS nodes fail, this can be tolerated: In case of a symmetric failure the failed TS node will be disregarded by all others, and these will synchronize to the then second fastest clock source, as before. In case of a failing clock source, the associated TS node will even remain synchronized (thanks to the relay rule), and all replica in the TMR are still operative. In case of an asymmetric fault (the individual TS nodes’ perceptions on whether a failure occurred differ in case of a broken communication link between two TS nodes, e.g.), the fully connected communication architecture, in combination with the algorithm design will still ensure synchrony among the clock outputs. This is actually the reason why we have \( 3f + 1 \) TS nodes (as for Byzantine-tolerant consensus) rather than just \( 2f + 1 \) (as for conventional majority voting). For details see [18].
Basically, the task of synchronizing clocks does not preclude the occurrence of glitches: An output clock that lagged behind might instantly reduce its phase lag through drastically shortening a half period (or more) without moving out of the skew tolerance. This violation of (R5) might indeed happen in the dynamic phase after some types of clock failure. This case will be illustrated in Section 6.3.

In the original DARTS this was not an issue since the operation of the TSA was assumed to be slow in general. However, with our assumptions (A2) and (A3) special provisions are needed. In particular, we need to artificially delay the delivery of the clock transitions to the inputs of the (A2) and (A3) special provisions are needed. In particular, we need to TSA was assumed to be slow in general. However, with our assumptions occurrence of glitches: An output clock that lagged behind might large white block below the pipeline is the compare unit that evaluates

5.1. Model and preliminaries

In the following.

5.2. Correctness analysis

Recall that \([\Delta_{\text{min}}, \Delta_{\text{max}}]\) is called the range of remote delay between nodes and \([\delta_{\text{min}}, \delta_{\text{max}}]\) is called the range of local delays within a node. The following observation shows that the names are justified within our model:

Observation 1. Let \(k \in \mathbb{N}^+\) and nodes \(i, j \in [n]\). Assume that correct node \(i\) sends tick \(k\) at time \(t\) and correct node \(j\) receives remote tick \(k\) from node \(i\) at time \(t\). Then, \(t' - t \in [\Delta_{\text{min}}, \Delta_{\text{max}}]\).

It follows from the definition of sending and receiving a tick and the semantics of the after-statement. Analogous statements hold for local ticks and oscillator ticks.

a) Design constraints. To show correctness of the algorithm we need to assume that certain design constraints hold. In particular we will assume:

\[
\delta_{\text{max}} < 2\delta_{\text{min}} \quad (1)
\]

\[
\delta_{\text{max}} \leq \Delta_{\text{min}} \quad (2)
\]

\[n \geq 3f + 1 \quad (3)
\]

For the moment we also assume that local and remote counter thresholds are infinite. We will later show that one can drop the assumption and assume finite (small) counter thresholds. Note that we do not assume infinite oscillator counter thresholds; oscillator ticks may thus be lost.

b) Correctness: We start the analysis by an observation on the generation of new ticks.

Observation 2. A correct node \(i \in [n]\) sends tick \(k \in \mathbb{N}^+\) at time \(t \geq 0\) only because either progress rule\(_i\) becomes true at time \(t\) or because relay rule\(_i\) becomes true at time \(t\).

Let \(U \subseteq [n]\). Motivated by Observation 2 we say the tick sent by node \(i\) at time \(t\) was based on tick-pairs \((r_u, e_u)\) from node \(u, u \in U, i, f\).

1) node \(i\) has received exactly \(r_u\) remote ticks from node \(u\) and exactly \(e_u\) local ticks for node \(u\), for all nodes \(u \in U\),

2) the counters of node \(i\) corresponding to nodes \(u \in U\) are such that progress rule\(_i\) or relay rule\(_i\) became true at time \(t\).

Note that the tick-pairs and nodes that a tick is based on are not necessarily unique.

Next, observe that for the polarities of ticks, even 0 and odd 1, the following properties hold:

Lemma 1. If a correct node \(i \in [n]\) sends tick \(k \in \mathbb{N}^+\) at time \(t \geq 0\), then:

1) The polarity of the tick, i.e., \(k \mod 2\), is equal to the value of cur_pol\(_i\) just before time \(t\).

2) The polarity of the tick is opposite to the value of next_pol\(_i\), for all nodes \(u\) the tick is based on.

If a correct node \(i \in [n]\) receives tick \(k \in \mathbb{N}\) from node \(j \in [n]\) at time \(t \geq 0\), then:

3) The polarity of the tick is opposite to the value of next_pol\(_j\) at time \(t\).

Proof. The first statement follows from the fact that cur_pol\(_i\) is initialized to 0 and toggled whenever node \(i\) sends a tick.

Likewise the third statement follows from the fact that next_pol\(_j\) is
initialized to 1 and toggled whenever node $i$ receives a tick from node $j$.

The second statement follows from the first statement, Observation 2, and the definition of progress rule and relay rule, as well as the definition of a tick being based on tick-pairs and nodes.

We are now in the position to show:

**Lemma 2.** If correct node $i \in [n]$ sends tick $k + 1 \in \mathbb{N}^+$ at time $t \geq 0$, then there exists a set of nodes $U \subseteq [n]$, such that the tick is based on tick-pairs $(r_{uw}, k)$ from node $u$, for $u \in U$. Further, the smallest time between successive ticks that a certain correct node sends is $\delta_{\min}$.

**Proof.** First observe that a tick $k + 1 \geq 1$ can only be based on local ticks $k, k - 2, k - 2, k - 4, k - 4, \ldots$, etc., because the polarity of such local ticks must be different to the polarity of tick $k + 1$ because of Lemma 1. Further, tick $k + 1$ cannot be based on local ticks greater than $k + 1$ since these ticks need to be sent before they are locally received. It follows that tick $k + 1$ must be based on local ticks within the set $(k, k - 2, k - 4, \ldots, 0)$.

The proof is by induction on $k + 1 \in \mathbb{N}^+$.

**Basis ($k + 1 = 1$):** By the above arguments, the tick $k + 1 = 1$ sent by a correct node $i$ must be based on local ticks $k = 0$; which proves the induction basis. Further, by the above observations, tick 2 can only be based on local tick 1. Thus, the minimum time between sending tick 1 and tick 2 is at least $\delta_{\min}$, the minimal delay between sending a local tick and receiving it.

**Step ($k + 1 \rightarrow k + 2$):** Assume that all ticks $k + 1 \geq 1$ sent by correct nodes were based on local tick $k \geq 0$. Consider a correct node $i \in [n]$ and let $t_f$ be the time it sent tick $\ell \in \mathbb{N}^+$. Recall, that tick $k + 1$ must be based on local ticks $k, k - 2, k - 4, \ldots$. The time tick $k - 2$ and all smaller ticks were sent by node $i$ is at most $t_k - 2\delta_{\min}$. However, all these ticks must have been received locally at the node by time $t_k - 2\delta_{\min} < t_k$.

the latter of which holds by Constraint (1). It follows that tick $k + 1$ sent by node $i$ can only be based on local tick $k$. Thus, the minimum time between sending tick $k + 1$ and tick $k$ is at least $\delta_{\min}$, the induction step follows.

**Lemma 3.** The first correct node $i \in [n]$ that sends tick $k + 1 \in \mathbb{N}^+$ does so because its progress rule, becomes true.

**Proof.** Assume by means of contradiction that this is not the case. By Observation 2, it must have sent tick $k + 1$ because relay rule, became true. From Lemma 2 we have that it must be based on local ticks $k$. But then from relay rule, for at least $(f + 1) - f = 1$ remote node $j \in [n]$ we have that $c_{ij} > 0$. For the node $j$, it is $c_{ij} = r_{ij} - k > 0$, by the assumption that counter thresholds are infinite and thus pipes do not loose ticks. It follows that correct node $j$ sent tick $r_{ij} < k$ before; a contradiction to the fact that node $i$ was the first correct one to send tick $k$.

**Lemma 4.** Let $t_{k-1}$ and $t_{k-1}'$ be the earliest times a correct node sends tick $k$ and $k + 1$, respectively. Then $t_{k-1} - t_k \geq \delta_{\min}$.

**Proof.** By Lemma 3, the first correct node that sends ticks $k + 1 \geq 1$ at time $t_{k-1} \geq 0$ must do so because its progress rule, becomes true. From Lemma 2 we have that it must be based on local ticks $k$. But then from progress rule, for at least $(2f + 1) - f = 1$ correct remote nodes $j \in [n]$ we have that $c_{ij} > 0$. For these nodes $j$, it is $c_{ij} = r_{ij} - k > 0$, by the assumption that counter thresholds are infinite and thus pipes do not loose ticks. Thus, at least $f + 1$ correct nodes must have sent tick $k$ at latest by time $t_{k-1} - \delta_{\min}$. It follows that $t_{k} \leq t_{k-1} - \delta_{\min}$.

We are now in the position to show that nodes cannot send ticks too far from each other.

**Lemma 5.** If the first correct node sends tick $k + 1 \geq 2$ at time $t_{k-1} \geq 0$ then all correct nodes send tick $k$ by time $t_{k-1} + \Delta_{\min} - \Delta_{\min} + \delta_{\max}$.

**Proof.** Set $T = \delta_{\max}$ in the following. We start with an observation: Assume that the first correct node, say node $i$, sends tick $k + 1 \geq 2$ at time $t_{k-1}$. By the same arguments as in the proof of Lemma 4, at least $f + 1$ correct nodes, say nodes $U \subseteq [n]$, must have sent tick $k + 1$ at latest by time $t_{k-1} + \Delta_{\min}$. Thus all correct nodes will receive tick $k$ from correct nodes in $U$ by time $t_{rem,k} = t_{k-1} + \Delta_{\min} + \delta_{\max}$.

We will now show that lemma’s statement by induction on $k + 1 \geq 2$.

**Basis ($k + 1 = 2$):** From the above observations and the fact that remote ticks are not lost by overflow, $c_{ij} u \geq k + 1$ at time $t_{rem,k} \leq t_{rem,2 + \delta_{\min} + \delta_{\max}}$ for all nodes $u \in U$. Since, further initially $curr_{pol,k} = 0 = -next_{pol,k}$, all for all $u \in U$ at time $t_{rem,2 + \delta_{\min} + \delta_{\max}}$, unless node $i$ has already sent tick 1 by this time, predicate relay rule holds for node $i$ before time $t_{k-1} + \Delta_{\max} - \Delta_{\min} + T$.

and it sends tick $k + 1$ by that time if it has not already done so. The lemma follows.

**Step ($k + 1 \rightarrow k + 2$):** As the induction hypothesis assume that the statement holds for $k + 1 \geq 2$. Now assume that the first correct node, say node $j$, sends tick $k + 2 \geq 3$ at time $t_{k-2}$. By the above arguments we have that by time $t_{k-1} + \Delta_{\max} - \Delta_{\min} + T + \delta_{\max}$ all correct nodes will receive tick $k + 1$ from a set $U' \subseteq [n]$ of at least $f + 1$ correct nodes.

Further, by the induction hypothesis, node $j$ has sent tick $k$ by time $t_{j-1} + \Delta_{\max} - \Delta_{\min} + T$.

Thus it will receive local tick $k$ for all nodes by time $t_{loc,k} = t_{k-1} + \Delta_{\max} - \Delta_{\min} + T + \delta_{\max}$.

Since neither remote nor local received ticks are lost due to the assumption of infinite counter thresholds, we have that at time $t_{loc,k} = t_{k-1} + \Delta_{\max} - \Delta_{\min} + T + \delta_{\max}$

max$(t_{rem,k}, t_{loc,k}) = \Delta_{\min} + \max(t_{rem,k} - t_{loc,k} + T + \delta_{\max}) \leq \Delta_{\max} - \Delta_{\min} + \max(t_{rem,k} - t_{loc,k} + T + \delta_{\max}) \leq \Delta_{\max} - \Delta_{\min} + \max(0, T) \leq t_{loc,k} + \Delta_{\max} - \Delta_{\min} + \delta_{\max} + \delta_{\max}$

for all nodes in $u \in U'$, it is $c_{ij} u \geq k + 1$ and $c_{ij} u = k$, unless node $j$ has already sent tick $k + 1$ by that time. It will thus send tick $k + 1$ by that time, unless it already did so. The induction step and thus the lemma follows.

The bound in Lemma 5 can be directly applied to infer how far apart correct nodes may produce ticks: it shows that fast nodes sending a tick $k + 1$ pull slow nodes behind them, making them send tick $k$. However, it does not yet allow us to infer a skew bound, i.e., a maximum duration between two ticks of the same number sent by two correct nodes.

Another problem not yet attacked is that it remains to show that correct nodes do not eventually deadlock, i.e., stop producing ticks.

The following lemma is key to answer the above two question:

**Lemma 6.** If all correct nodes send tick $k \in \mathbb{N}^+$ by time $t$, then all correct nodes send tick $k + 1$ by time $t + \max(\Delta_{\max} 4\Pi_{\max})$.

**Proof.** Assume that all correct nodes send tick $k \in \mathbb{N}^+$ by time $t$. Then all correct nodes will receive remote tick $k$ from all correct nodes, that is, because of (3), at least $n - f \geq 2f + 1$ many, by time $t + \Delta_{\max}$. They will also receive local tick $k$ for these nodes by time $t + \delta_{\max}$. Further, at time $t$ it must have been the case that for a correct node $i$ it is $cl_{i,\Pi_{\max}} \geq 1 - n$ by the fact that the counter is bounded from below. Thus after at most another $3\Pi_{\max}$ time, it is $cl_{i,\Pi_{\max}} \geq 1$, if node $i$ has not already sent tick $k + 1$. At most another $\Pi_{\max}$ later, the value of $next_{pol,\Pi_{\max}}$ is set to $k + 1$ mod 2, i.e., as required by progress rule for node $i$ to send tick $k + 1$.

Combining the above, we obtain that progress rule holds by time $t + \max(\Delta_{\max}; \Delta_{\max}; 4\Pi_{\max})$. \hfill (4)
if node $i$ has not already sent tick $k + 1$ before; making it send tick $k + 1$ by that time. The lemma follows.

Combining Lemma 4 and Lemma 5 we finally obtain our main result showing synchrony of generated ticks and the possibility to use (small) bounded counters $c_{i,j}$. We may thus drop our initial assumption that $c_{i,j}$ are unbounded.

**Theorem 1.** For all correct nodes $i, j \in [n]$:

1) The counter $c_{i,j}$ remains within a bounded range at all times and thus does not underrun or overrun.

2) Nodes $i$ and $j$ send tick $k$ within a bounded time range.

3) Node $i$ never deadlocks.

**Proof.** To show the first two bounds we use Lemma 5 to infer that two correct nodes $i, j \in [n]$ where $i$ is among the first to send tick $k + 1$ and $j$ among the last to send tick $k$, send their respective ticks $k + 1$ and $k$ within time $\Delta_{\text{max}} - \Delta_{\text{min}} + \delta_{\text{max}}$.

From Lemma 6 we have that node $j$ will send tick $k + 1$ another time $\max(\Delta_{\text{max}}, 4\Pi_{\text{max}})$ later. Thus the maximum difference in time between any two correct nodes sending tick $k + 1$, i.e., the maximum skew, is $\Delta_{\text{max}} - \Delta_{\text{min}} + \delta_{\text{max}} + \max(\Delta_{\text{max}}, 4\Pi_{\text{max}})$.

The fact that counters $c_{i,j}$ remain bounded for correct nodes $i, j \in [n]$ follows by the fact that any two correct nodes send tick $k + 1$ within bounded time of each other, and the fact that the first correct node can produce new ticks with a period of at least $\Delta_{\text{min}}$ during this time by Lemma 4.

The last statement follows from the observation that all correct nodes will eventually send tick 1 and repeated application of Lemma 6.

6. **Experimental evaluation**

To illustrate our approach we show selected results from the numerous simulation runs we have performed for its validation. These simulations were all performed on a digital abstraction level by using QuestaSim for simulating our VHDL (pre-layout) implementation that allows to freely adjust the delays for the delay line and the interconnect between TS nodes with one picosecond granularity. We use LROs with significantly different frequency in these simulations to allow for a better distinction among them. In practice one would of course choose well matched reference clocks to obtain the best possible accuracy.

6.1. **Steady state operation**

Fig. 2 illustrates the steady state behavior of the algorithm as already outlined in the previous section. It can be verified that LROs dictate the timing, and all clock outputs follow its frequency, while the other LROs get out of sync.

Let us investigate the flow of events in more detail: As soon as LRO2 fires, TS2 can fire as well, after having processed the LRO transition, which takes $\Delta_{A, 2}$. The transition thus produced by TS2 goes to all TS nodes (with transmission delay $\Delta_{T, 23}$) where it is processed. Specifically it will cause the firing of TS4 and TS8 through the relay rule. So these clocks will be delayed against TS2 by $\Delta_{T, 23} + \Delta_{A, 3}$ and $\Delta_{T, 24} + \Delta_{A, 4}$, respectively. Their firing, in turn, will make TS1 execute its increment rule, as soon as the earlier one of these two transitions is received. So after the firing of TS2 we have two concurrent paths, the faster of which will cause the firing of TS1: The path over TS2 comprises $\Delta_{T, 23} + \Delta_{A, 3} + \Delta_{T, 31}$, and the other one $\Delta_{T, 24} + \Delta_{A, 4} + \Delta_{T, 41}$. The firing of TS1 actually starts the next round; it is a transition of inverted polarity. This puts TS1 nearly one half period II ahead of TS2; precisely

![Fig. 2. Simulation trace for the fault-free steady state operation.](image1)

![Fig. 3. Simulation trace showing the failure of one clock source.](image2)
which is essentially one half period minus two interconnect delays and two processing delays (2(Δt2 + Δλ)). The arrival of TS2 and TS3’s transition had also armed TS2 to fire its increment rule, but this TS node still needs to wait for its LRO transition to arrive. When this occurs, TS2 fires and the cycle starts over.

6.2. Failure of the fastest TS node

The situation shown in Fig. 3 represents a very unfavorable case: When TS1 loses its LRO due to a failure (we assume a crash failure, i.e., it stops oscillating), LRO2 replaces it as the fastest TS node, and LRO3 becomes the dominating source. This means that from this moment on TS3 is no longer “pulled” by the relay rule, but needs to wait for its increment rule to fire. This, in turn, requires it to wait for transitions from its LRO. While these transitions do arrive, the first one of these gets canceled through our difference counter (recall that the input pipe carries already two entries). As a result, the output clock will not proceed until two more transitions from the LRO have arrived. This will cause the clock to stall for a while. We believe that this is an inevitable price to be paid for synchronization – after all we need to accomplish the switch-over from LRO2 which was dominating so far, to LRO3 as the dominant one, which is, however, out of sync at the moment of switching. At this point it also becomes clear why the buffer should not be deeper than two transitions. The maximum pulse width \( H_{\text{max}} \) can be determined as follows:

\[
H_{\text{max}} = \Delta t_{31} + 3 \Delta t_{13} + \Delta t_{41} + \Delta t_{23} + \Delta t_{31} + \Delta t_{24} + \Delta t_{41} + \Delta t_{13}
\]

6.3. Glitch failure

Another unfavorable scenario is depicted in Fig. 4. Here we assume that TS2 produces a glitch due to a failure and then stops operating. The second, early, transition of this glitch, together with the early transition had also armed TS2 to fire its increment rule, but this TS node still needs to wait for its LRO transition to arrive. When this occurs, TS2 fires and the cycle starts over.

6.4. Changing frequency

Although the primary application of our approach will most likely be tolerating the crash failure of a crystal, it is interesting to see what happens in case of significant frequency changes in the LRO. This is illustrated in Fig. 5: On the left side of the trace the two second fastest crystal, LRO2, speeds up and its associated node TS2 becomes the fastest node. As a consequence, TS1 gets the role of the second fastest node and hence its timing dominates the output, starting from \( t = 1645 \text{ns} \) (see marker). As can be seen, there is only little effect on the generated clocks, apart from LRO2 managing to speed up its associated clock output a bit – but no pulse gets shorter than \( \delta_{\text{LRO}} \). It can also be seen that it takes a couple of clock periods for TS1 to take over the lead from TS2; this is because of the transitions buffered in the elastic pipelines.

In the same figure, further to the right, we have the case that LRO2 suddenly slows down, now becoming the slowest clock. The end result is as expected: TS2 takes over the role of the second fastest, leading node, as can be seen from the marker at 2255ns onward. The process of getting there, however is interesting: Due to the buffering of transitions, LRO2’s change in speed is not immediately reflected in the clock outputs. In a first phase TS2 moves from the fastest to the second fastest position (the LRO transitions stored in its LRO pipeline are used up), and consequently takes the lead for some time, precisely from \( t = 1975 \text{ns} \) to \( t = 2255 \text{ns} \) (see markers). Later on (when the LRO pipeline starts containing negative entries), TS2 loses the lead to TS3 and goes to the set of slowest nodes, together with TS4. So again we observe a transient phase during which the clock is not perfectly stable, before the output frequency stabilizes again.

The duration of these transient phases depends on (a) buffer depth and (b) frequency mismatch. Considering (a) we have used minimum buffer depth. With respect to (b) we can envision two extremes: In case the frequency change seen at one node (here it was TS2) is large, the buffers will be adapted fast and the transient phase be short – but exhibit a potentially significant change in the output frequency. In case of a relatively minor frequency change we will observe a long transient phase that, however, causes little change in the output frequency.

From the observations made so far we can conclude that jitter in the LRO will be directly reflected as jitter in the output clock, as long as the dominating source stays the same. For very well matched LRO frequencies, jitter might cause the dominant LRO to change frequently. However, in that case, the respective LRO counters won’t overrun or underrun (due to the good frequency matching), so the hand-over from one leader to another will not cause significant transient phases at the output. In any case, LRO jitter will, in general, not be mitigated by our approach, but jitter will not invalidate our approach either.
symmetry, typically below one clock cycle). Disruptive events, however, may lead to a hand-over from one LRO to another one, which entails a minor frequency change (as much as the tolerance between the clock sources amounts to), as well as possibly a short idle period. Normally, this will not cause any problem to the operation of the connected nodes (with clock gating, e.g., they will see a much larger idle period; also output, with a bounded mutual phase shift (depending on routing and supplies multiple hardware modules in a chip (like the replica of a TMR architecture, e.g.) with clocks that are synchronized and still fail independently. This becomes challenging as soon as these clocks shall exhibit the good accuracy and stability of a crystal oscillator. The approach we presented builds on the tick synchronization algorithm by Srikanth and Toueg [18], as refined in the distributed clock generation scheme DARTS whose tolerance against Byzantine failures has been formally proven in previous works. We extend this approach by augmenting it with stable reference clocks to improve its accuracy and stability. We have formally specified the new algorithm and illustrated its operation principle by detailed explanations and simulation examples. Furthermore, we have given a formal correctness proof for the new algorithm.

We have formulated a list of requirements that we deem desirable for the intended use of the approach. Among these are tolerance of single faults in the crystal oscillators as well as the circuit implementing the algorithm, absence of any single point of failure, a lower limit for the pulse width (no glitching), as well as an accuracy comparable to that of the crystal oscillators. It turns out that our solution meets all these requirements, with the exception of continuous stability of the output clocks: During stabilization phases after some specific types of failure, there may be shorter or longer clock periods. We have illustrated that through characteristic simulation results. The limits for both cases have been well specified in our proof, and all demands of the supplied circuit and the executed application will typically be completely fulfilled. It seems that these residual temporary limitations are the price to be paid for synchronizing independent clock sources, as perfectly following a single stable clock source and at the same time, once that source fails, seamlessly switching over to a fully independent redundant one without any phase jumps – as continuous stability would imply –, is not feasible in principle.

6.5. Discussion

Our experiments have in general confirmed our hope that, based on the DARTS approach, synchronized clocks can be provided to a set of nodes (like the redundant nodes in a TMR architecture), without accepting the clock as a single point of failure. Most of the time, namely during fault-free and stable operation of LROs and TS nodes, the produced clocks follow the second fastest LRO consistently on all clock outputs, with a bounded mutual phase shift (depending on routing symmetry, typically below one clock cycle). Disruptive events, however, may lead to a hand-over from one LRO to another one, which entails a minor frequency change (as much as the tolerance between the clock sources amounts to), as well as possibly a short idle period. Normally, this will not cause any problem to the operation of the connected nodes (with clock gating, e.g., they will see a much larger idle period; also clock failure detection and switch-over to a redundant clock entails a longer idle time due to the necessary synchronizers), unless the clock is used for some kind of real-time sampling. Most importantly, the clock outputs stay mutually synchronized even throughout these phases.

In exceptional cases a clock pulse could even become shorter than that of the fastest LRO. This can be mitigated by matching the delay element $δ_{\text{LRO}}$ in the local feedback path appropriately. Without this provision, in case of a substantial mismatch, the node, which was designed to operate with the LRO clock, might be upset by the short pulse.

It should be noted here, that the simulations alone cannot prove the validity of our approach. That is why we have provided a formal proof in Section 5 that clearly states the conditions under which the approach will work and gives the formal underpinning and guarantees. In this sense the simulations just serve as an illustration and sanity check for important scenarios.

7. Conclusion

We have motivated the need for a clock generation approach that supplies multiple hardware modules in a chip (like the replica of a TMR architecture, e.g.) with clocks that are synchronized and still fail independently. This becomes challenging as soon as these clocks shall furthermore exhibit the good accuracy and stability of a crystal oscillator. We have formulated a list of requirements that we deem desirable for

CRediT authorship contribution statement

Andreas Steininger: Conceptualization; Funding acquisition; Investigation; Methodology; Project administration; Resources; Writing - original draft; Writing - review & editing.
Wolfgang Duer: Data curation; Investigation; Validation; Visualization; Writing - original draft; Writing - review & editing.
Matthias Fuegger: Conceptualization; Formal analysis; Investigation; Methodology; Writing - original draft; Writing - review & editing.

Contributions

This paper is a substantial extension of our paper “Merging Redundant Crystal Oscillators into a Fault-Tolerant Clock” from DDECS 2020 where it received the best paper award in the category ‘Digital Design’.
This extension is a formal description of the algorithm, a formal proof of its proper function, the necessary boundary conditions (delays, etc) and its essential properties. In addition we have extended the experimental results presented, as well as revised the overall presentation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been partly supported by the Intel Collaborative Research Institute “Collaborative Autonomous and Resilient Systems”, by DigiCosme (working group HicDiesMeus), and by the INS2I.
Émergence grant BACON. The authors furthermore acknowledge TU Wien Bibliothek for financial support through its Open Access Funding Programme.

References


