Generation of a fault-tolerant clock through redundant crystal oscillators

Wolfgang Dür, Matthias Függer, Andreas Steininger

Generation of a Fault-Tolerant Clock through Redundant Crystal Oscillators

Wolfgang Dür*, Matthias Függer† and Andreas Steininger*
*TU Wien, Institute of Computer Engineering
A-1040 Vienna, Austria
{wduer,steininger}@ecs.tuwien.ac.at
†CNRS & LMF, ENS Paris-Saclay, Université Paris-Saclay & Inria
France
mfuegger@lsv.fr

***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.

I. 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 II we will briefly review and discuss existing approaches, before elaborating the requirements we want a solution to meet in Section III. Subsequently, we will briefly introduce the DARTS approach that our solution builds upon, along with the extensions we propose, in Section IV. Next, Section V 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 VI, before we conclude the paper in Section VII.

II. 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], [2], [3], [4], [5], [6], [7], [8], [9]. This method is, e.g., also used in Intel's STRATIX10 FPGAs[1]. The challenges with this approach are (a) to avoid glitching 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

[1]Intel STRATIX 10 Clocking and PLL User Guide https://www.intel.com/content/www/us/en/programmable/documentation/mcn1440569668630.html

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], [15], [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 IV 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.

III. 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_{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).

IV. Proposed Solution for Redundant Clock Outputs

A. Starting point: The DARTS approach

The DARTS architecture (**D**istributed **A**lgorithm for **R**obust **T**ick **S**ynchronization) implements the formally proven fault-tolerant tick synchronization algorithm (TSA) from Srkianth 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 interconnected. 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).

B. 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.
- **(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. 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.

C. 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. 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 1. More specifically, lines 1...5 express how, by the operator $counter \oplus^{[\delta_a,\delta_b]} (i, j)$, 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).

[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.

Algorithm 1

1: **define** $counter \oplus^{[\delta_a,\delta_b]} (i,j)$
2: **if** $c_{i,j} < 3$ **then**
3: $c_{i,j} \leftarrow c_{i,j} + 1$ after $[\delta_a, \delta_b]$
4: **end if**
5: **end define**
6:
7: **define** $counter \ominus^{[\delta_a,\delta_b]} (i,j)$
8: **if** $c_{i,j} > -2$ **then**
9: $\left\{ \begin{array}{l} c_{i,j} \leftarrow c_{i,j} - 1 \\ next\_pol_{i,j} \leftarrow \neg next\_pol_{i,j} \end{array} \right\}$ after$[\delta_a, \delta_b]$
10: **else**
11: $next\_pol_{i,j} \leftarrow \neg next\_pol_{i,j}$ after $[\delta_a, \delta_b]$
12: **end if**
13: **end define**
14:
15: **define** $counter_{LRO} \oplus^{[\delta_a,\delta_b]} (i)$
16: **if** $c_{LRO_i} < 2$ **then**
17: $c_{LRO_i} \leftarrow c_{LRO_i} + 1$ after $[\delta_a, \delta_b]$
18: **end if**
19: **end define**
20:
21: **define** $counter_{LRO} \ominus^{[\delta_a,\delta_b]} (i)$
22: **if** $c_{LRO_i} > -2$ **then**
23: $\left\{ \begin{array}{l} c_{LRO_i} \leftarrow c_{LRO_i} - 1 \\ next\_pol_{LRO_i} \leftarrow \neg next\_pol_{LRO_i} \end{array} \right\}$ after$[\delta_a, \delta_b]$
24: **else**
25: $next\_pol_{LRO_i} \leftarrow \neg next\_pol_{LRO_i}$ after $[\delta_a, \delta_b]$
26: **end if**
27: **end define**
28:
29: **define** $LRO\_tick(i)$
30: wait for time $t \in [\Pi_{i_{min}}, \Pi_{i_{max}}]$
31: $counter_{LRO} \oplus^{[\Delta_{i_{min}},\Delta_{i_{max}}]} (i)$
32: **end define**
33:
34: **variables**
35: cur_pol_i : bool $\leftarrow 0$ $(i \in P)$
36: $next\_pol_{i,j}$: bool $\leftarrow 1$ $(i,j \in P)$
37: $next\_pol_{LRO_i}$: bool $\leftarrow 1$ $(i \in P)$
38: $c_{i,j}$: integer $\leftarrow 0$ $(i,j \in P)$
39: c_{LRO_i} : integer $\leftarrow 0$ $(i \in P)$
40: **end variables**
41:
42: **foreach** $i \in P$ **do repeatedly in parallel**
43: $ready_{LRO_i} \leftarrow cur\_pol_i \neq next\_pol_{LRO_i} \wedge c_{LRO_i} > 0$
44: $\forall j \in P : rdyrelay_{rem_{i,j}} \leftarrow cur\_pol_i \neq next\_pol_{i,j} \wedge c_{i,j} > 0$
45: $\forall j \in P : rdyprogress_{rem_{i,j}} \leftarrow cur\_pol_i \neq next\_pol_{i,j} \wedge c_{i,j} \geq 0$
46: $progress\_rule_i \leftarrow (\sum_{j \in P} rdyprogress_{rem_{i,j}}) \geq 2f + 1 \wedge ready_{LRO_i}$
47: $relay\_rule_i \leftarrow (\sum_{j \in P} rdyrelay_{rem_{i,j}}) \geq f + 1$
48:
49: **if** $progress\_rule_i \vee relay\_rule_i$ **then**
50: $cur\_pol_i \leftarrow \neg cur\_pol_i$
51:
52: $\forall j \in P : counter \ominus^{[\delta_{i_{min}},\delta_{i_{max}}]} (i,j)$
53: $counter_{LRO} \ominus^{[\delta_{i_{min}},\delta_{i_{max}}]} (i)$
54:
55: $\forall j \in P : counter \oplus^{[\Delta_{min},\Delta_{max}]} (j,i)$
56: **end if**
57:
58: $LRO\_tick(i)$
59: **end foreach**

Similarly, lines 7...13 express how the $counter \ominus^{[\delta_a,\delta_b]} (i,j)$ operator decrements a counter through a local tick, this time using a cut-off value of -2. The important difference to the above operator can be found in lines 9 and 11: here the flag $next\_pol$ is toggled to acknowledge the reception of the local tick and by this prevent the algorithm from re-evaluating the same rules that just caused the firing of the tick over and over and in that way create multiple ticks (This will be detailed below). In hardware this is implemented by separating the evaluation of the firing conditions for rising and falling edges. It is important that the flag is not toggled before the counter has been decremented, therefore we apply the same "after" attribute, with the same choice of the delay within the given interval, to both assignments in line 9 at the same time.

Finally, lines 15...32 describe the maintenance of a counter for the local reference oscillator (LRO) associated with each TS node. The operators $counter_{LRO} \oplus^{[\delta_a,\delta_b]} (i)$ and $counter_{LRO} \ominus^{[\delta_a,\delta_b]} (i)$ are equivalent to their counterparts for the local tick counters explained above, albeit with different cut-off values of +2 and -2, respectively.

The LRO counter c_{LRO_i} is incremented by each transition of the reference oscillator through the operator $LRO\_tick(i)$. Note that this is accompanied by delaying the completion of the operation for one half period Π_i of the LRO clock through the $wait$ statement.

With these operators being defined, the actual algorithm can start with an initialization (lines 34...40): All counters $(c_{i,j}, c_{LRO_i})$ are initialized to zero, all clock outputs (cur_pol_i) to logic low, and the handshake flags $(next\_pol_i)$ to logic high.

Next, the counter states are evaluated: In line 43 the statement $c_{LRO_i} > 0$ 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 $ready_{LRO_i}$, the readiness for firing a new local tick. This, however only happens in case the handshake allows for that: The condition $cur\_pol_i \neq next\_pol_{LRO_i}$ 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 $next\_pol_{LRO_i}$ – and has not already been used for generating the previous transition that led to the level indicated by cur_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 ($rdyrelay_{rem_{i,j}}$) or more than 0 ($rdyprogress_{rem_{i,j}}$) 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 $rdyprogress_{rem_{i,j}}$ activated, the progress rule makes that TS node fire. However, the specific feature of our algorithm is that this does not happen without $ready_{LRO_i}$ 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 $rdyrelay_{rem}$ 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 δ is in the interval $[\delta_{i_{min}}, \delta_{i_{max}}]$ 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 Δ from the interval $[\Delta_{i_{min}}, \Delta_{i_{max}}]$ that is solely 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 cur_pol and $next\_pol$ ensures an appropriate interleaving of the concurrent executions.

In the same way the LRO clock expressed by the statement $LRO\_tick(i)$ 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.

D. 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 (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 TS_1 will have to wait for TS_2 and TS_3 before it gets released to fire. Even though TS_2 receives this rising transition, it still cannot fire: For executing the relay rule that one transition is too little (it needs at least 2); for the increment rule to work, it still needs to receive a transition from its LRO. When the transition from LRO_2 finally arrives, TS_2 will fire. Now TR_3 and TR_4, who had already received the rising transition from TS_1 also receive the one from TS_2, which is sufficient to fire without waiting for the LRO. Once the rising transitions thus generated by $TS3$ and $TS4$ are received by TS_1, the latter can execute its increment rule to arm its TSA for the next falling transition. Since LRO_1 had already delivered the required transition (and made $c_{LRO_1} > 0$) , this firing will happen immediately and the next cycle starts.

This cyclic process has several consequences:

- The speed of TS_2 will dictate the overall system speed. The clock outputs all follow f_2. For an illustration see Figure 2 in Section VI.
- As a consequence the fastest LRO, the one at TS_1, runs ahead and gets out of sync. We therefore need to buffer at least one of its transitions to make 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 TS_3, 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, like 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 VI-C.

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 TS nodes (by δ_H) to prevent premature firing and thus enforce that every half period is larger than H_{min}. Note that the choice of δ_H implies a compromise: Choosing δ_H too low will not yield the effect of getting above H_{min}, while too large a δ_H will invalidate (A3) and hence the described operation principle of the extended algorithm.

Figure 1 shows the proposed circuit implementation. The basic blocks and their construction are illustrated, with a special emphasis on the proposed changes to the DARTS implementation. The upper part of the figure shows one up/down counter with the ticks received at the left, while the produced tick is entering from the right. Each TS node comprises 5 of these counters (symbolically shown as "'shadows" in the background), one for each produced TS clock and one for the LRO. The large white block below the pipeline is the compare unit that evaluates the pipeline fill level ("greater" and "greater equal" for odd (rising) and even (falling) transitions).

The box at the bottom of the figure shows how the comparison results are processed by the TSA's rules: The rectangles with the conditions are actually threshold gates that implement these rules, two for the increment rule (for odd and even), and another two for the relay rule. Building such threshold gates in a conventional way by a sum-of-products implementation scales badly; a more efficient solution is by means of sorting networks. The threshold gates are supplied at their inputs with the respective comparison signals from all 5 counters. In comparison with the original DARTS implementation the AND gates have been added to make the increment rule firing conditional to the LRO's pipe fill level. The gates in the right part merge these rules appropriately, such that at the right side the generated tick leaves the block.

For a more in-depth discussion and reasoning about the implementation details of the original DARTS components, please refer to [16]. A formal proof of the DARTS algorithm as well as a derivation of the synchronization bounds can be found in [20] and [21].

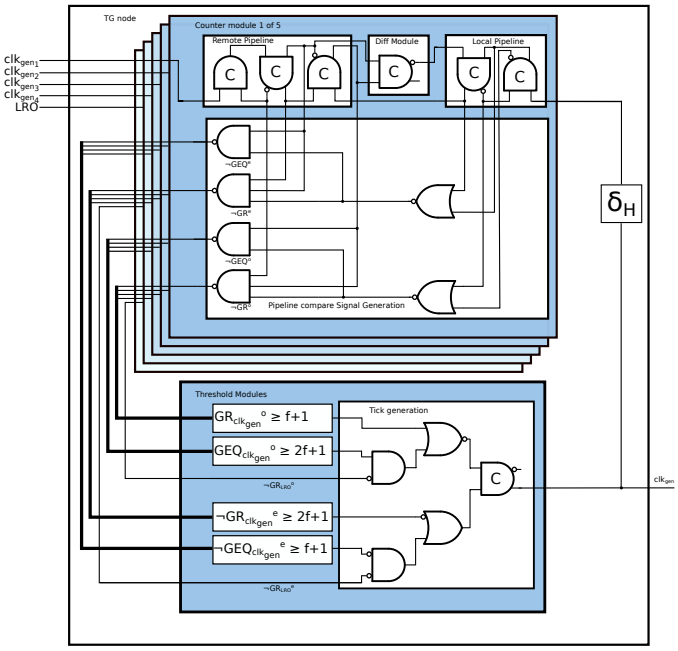

Fig. 1. Proposed TS node implementation

V. Formal Correctness Proof

As the correctness proofs from DARTS do not account for the proposed extension by crystal clock sources, new proofs need to be elaborated to guarantee the correctness of our approach. This will be the focus in the following.

A. Model and preliminaries

We denote the set of natural numbers by $\mathbb{N} = \{0, 1, \dots\}$ and write $\mathbb{N}^+ = \{1, 2, \dots\}$. Let $[n] = \{1, 2, \dots, n\}$ for $n \in \mathbb{N}$.

As laid out before we consider a system of $n \in \mathbb{N}^+$ nodes $P = [n]$ up to $f \in \mathbb{N}$ of which can be Byzantine. While correct nodes must adhere to the algorithm, i.e., execute the **foreach**-loop as stated, a faulty node i can manipulate its variables, i.e., $c_{i,\cdot}$, cur_pol_i, $next\_pol_{i,\cdot}$, and c_{LRO_i}, arbitrarily.

We say a node $i \in [n]$ *sends tick* $k \in \mathbb{N}^+$ *at time* $t \geq 0$ if the code within the **if**-statement in line 47 for node i is executed for the k^{th} time and this happens at time t. Node i *sends oscillator tick* $k \in \mathbb{N}^+$ *at time* $t \geq 0$ if line 29 with parameter i is executed for the k^{th} time and this happens at time t. A node i *receives remote tick* $k \in \mathbb{N}^+$ *from node* $j \in [n]$ *at time* $t \geq 0$ if $c_{i,j} \leftarrow c_{i,j} + 1$ (line 3) is executed for the k^{th} time and this happens at time t. Likewise, it *receives local tick* $k \in \mathbb{N}^+$ *for node* j *at time* $t \geq 0$ if $c_{i,j} \leftarrow c_{i,j} - 1$ (line 9) is executed for the k^{th} time and this happens at time t. Node i *receives oscillator tick* $k \in \mathbb{N}^+$ *at time* $t \geq 0$ if $c_{LRO_i} \leftarrow c_{LRO_i} - 1$ (line 22) is executed for the k^{th} time and this happens at time t. We say that all nodes receive local, remote, and oscillator tick 0 at time 0.

B. Correctness analysis

Recall that $[\Delta_{\min}, \Delta_{\max}]$ is called the range of remote delay between nodes and $[\delta_{\min}, \delta_{max}]$ 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, $t' - t \in [\Delta_{\min}, \Delta_{\max}]$.*

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:

$$\delta_{\max} < 2\delta_{\min} \tag{1}$$
$$\delta_{\max} \leq \Delta_{\min} \tag{2}$$
$$n \geq 3f + 1 \tag{3}$$

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, ℓ_u) from node u, for $u \in U$*, if

1) node i has received exactly r_u remote ticks from node u and exactly ℓ_u local ticks for node u, for all nodes $u \in U$, and
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 $next\_pol_{i,u}$ 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 $next\_pol_{i,j}$ 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 form the fact that $next\_pol_{i,j}$ 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 $progress\_rule_i$ and $relay\_rule_i$, 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_u, 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, $k - 2$, $k + 2$, $k - 4$, $k + 4$, 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 ($k + 1 \to k + 2$): 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_ℓ be the time it sent tick $\ell \in \mathbb{N}^+$. Recall, that tick $k + 1$ must be based on local ticks k, $k - 2$, or $k - 4$, etc. The time tick $k - 2$ and all smaller ticks were sent by node i is at most $t_k - 2\delta_{\min}$. However, thus, all these ticks must have been received locally at the node by time

$$t_k - 2\delta_{\min} + \delta_{\max} < t_k \ ,$$

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 $progress\_rule_i$ 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 $relay\_rule_i$ became true. From Lemma 2 we have that it must be based on local ticks k. But then from $relay\_rule_i$, for at least $(f+1)-f = 1$ remote node $j \in [n]$ we have that $c_{i,j} > 0$. For the node j, it is $c_{i,j} = r_{i,j} - k > 0$, by the assumption that counter thresholds are infinite and thus pipes do not loose ticks. It follows that correct node j sent tick $r_{i,j} > k$ before;

a contradiction to the fact that node i was the first correct one to send tick k. □

Lemma 4. *Let t_k 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 tick $k+1 \geq 1$ at time $t_{k+1} \geq 0$ must do so because its $progress\_rule_i$ becomes true. From Lemma 2 we have that it must be based on local ticks k. But then from $progress\_rule_i$, for at least $(2f+1) - f = f+1$ correct remote nodes $j \in [n]$ we have that $c_{i,j} > 0$. For these nodes j, it is $c_{i,j} = r_{i,j} - k \geq 0$, 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*

$$t_{k+1} + \Delta_{\max} - \Delta_{\min} + \delta_{\max} \ .$$

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 $k \geq 1$ 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, $c_{i,u} \geq k = 1$ at time $t_{rem,k} \leq t_{rem,k} + T$ for all nodes $u \in U$. Since, further initially $curr\_pol_i = 0 = \neg next\_pol_{i,u}$ for all $u \in U$ at time $t_{rem,k} + T$, unless node i has already sent tick 1 by this time, predicate $relay\_rule_i$ holds for node i before time

$$t_{k+1} + \Delta_{\max} - \Delta_{\min} + T \ ,$$

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

Step ($k+1 \to k+2$): 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

$$t_{rem,k+1} = t_{k+2} - \Delta_{\min} + \Delta_{\max}$$

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

$$t_{k+1} + \Delta_{\max} - \Delta_{\min} + T \ .$$

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

$$\begin{aligned}
&\max(t_{rem,k+1}, t_{loc,k}) = \\
&\Delta_{\max} - \Delta_{\min} + \max(t_{k+2}, t_{k+1} + T + \delta_{\max}) \overset{Lemma\ 4}{\leq} \\
&\Delta_{\max} - \Delta_{\min} + \max(t_{k+2}, t_{k+2} - \Delta_{\min} + T + \delta_{\max}) \overset{(2)}{\leq} \\
&t_{k+2} + \Delta_{\max} - \Delta_{\min} + \max(0, T) \leq \\
&t_{k+2} + \Delta_{\max} - \Delta_{\min} + T \ ,
\end{aligned}$$

for all nodes in $u \in U'$, it is $c_{j,u} \geq k+1$ and $c_{j,u} = k$, 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 k 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 $c_{LRO_i} \geq -1$ by the fact that the counter is bounded from below.[4] Thus after at most another $3\Pi_{\max}$ time[5], it is $c_{LRO_i} \geq 1$, if node i has not already sent tick $k+1$. At most another $\Pi_{\max}$ later, the value of $next\_pol_{LRO_i}$ will be set to $k+1 \mod 2$, i.e., as required by $progress\_rule_i$ for node i to send tick $k+1$.

Combining the above, we obtain that $progress\_rule_i$ holds by time

$$\begin{aligned}
&t + \max(\Delta_{\max}, \delta_{\max}, 4\Pi_{\max}) \overset{(2)}{=} \\
&t + \max(\Delta_{\max}, 4\Pi_{\max}) \ ,
\end{aligned}$$

if node i has not already sent tick $k+1$ before; making it send tick $k+1$ by that time. The lemma follows. □

[4] In fact we even have $c_{LRO_i} \geq 1$ if node i sent tick k via $progress\_rule_i$.

[5] The next increment may happen $\Pi_{\max}$ after time t if an increment happened just before time t.

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 $\Delta_{\max} - \Delta_{\min} + \delta_{\max}$.

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

$$\Delta_{\max} - \Delta_{\min} + \delta_{\max} + \max(\Delta_{\max}, 4\Pi_{\max}) \ .$$

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 $\Delta_{\min}$ 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. □

VI. 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[6]) 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.

A. Steady state operation

Figure 2 illustrates the steady state behavior of the algorithm as already outlined in the previous section. It can be verified that LRO_2 dictates 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 LRO_2 fires, TS_2 can fire as well, after having processed the LRO transition, which takes $\Delta_{A,2}$. The transition thus produced by TS_2 goes to all TS nodes (with transmission delay $\Delta_{T,2i}$) where it is processed. Specifically it will cause the firing of TS_3 and TS_4 through the relay rule. So these clocks will be delayed against TS_2 by $\Delta_{T,23} + \Delta_{A,3}$ and $\Delta_{T,24}+\Delta_{A,4}$, respectively. Their firing, in turn, will make TS_1 execute its increment rule, as soon as the earlier one of these two transitions is received. So after the firing of TS_2 we have two concurrent paths, the faster of which will cause the firing of TS_1: The path over TS_3 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 TS_1 actually starts the next round; it is a transition of inverted polarity. This puts TS_1 nearly one half period Π ahead of TS_2; precisely

$$\Pi_2 - (\min(\Delta_{T,23}+\Delta_{A,3}+\Delta_{T,31}, \Delta_{T,24}+\Delta_{A,4}+\Delta_{T,41}) + \delta_{A,1}), \tag{4}$$

which is essentially one half period minus two interconnect delays and two processing delays ($2(\Delta_T + \Delta_A)$). The arrival of TS_3 and TS_4's transition had also armed TS_2 to fire its increment rule, but this TS node still needs to wait for its LRO transition to arrive. When this occurs, TS_2 fires and the cycle starts over.

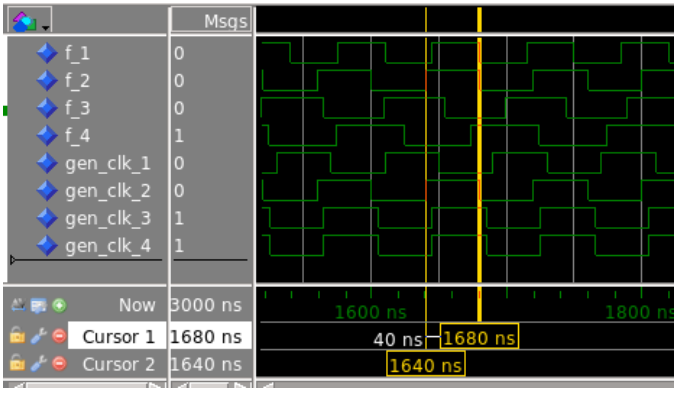

Fig. 2. Simulation trace for the fault-free steady state operation

B. Failure of the fastest TS node

The situation shown in Figure 3 represents a very unfavorable case: When TS_1 loses its LRO due to a failure (we assume a crash failure, i.e., it stops oscillating), LRO_2 replaces it as the fastest TS node, and LRO_3 becomes the dominating source. This means that from this moment on TS_3 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 LRO_2 which was dominating so far, to LRO_3 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 $H_{\max}$

[6]Since DARTS had been implemented in an ASIC [16], it is evident that our extension (that essentially adds an OR gate as well as a couple of further Muller-pipelines) can be synthesized as well. Therefore this question has not been in the focus of our design.

can be determined as follows:
$H_{\max} = \Delta_{T,31} + 3\Pi_3 + \Delta_{A_3} + \Delta_{T,23} + \Pi_2 - \min(\Delta_{T,23} + \Delta_{A_3} + \Delta_{T,31}, \Delta_{T,24} + \Delta_{A_4} + \Delta_{T,41})$

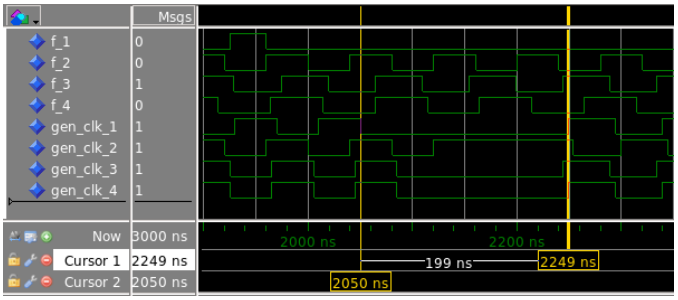

Fig. 3. Simulation trace showing the failure of one clock source

C. Glitch failure

Another unfavorable scenario is depicted in Figure 4. Here we assume that TS_2 produces a glitch due to a failure and then stops operating. The second, early, transition of this glitch, together with the early transition of TS_1 (which is part of the regular behavior, as illustrated above) will unduly fire the relay rules of TS_3 and TS_4, thus propagating the glitch. In this case, the pulse width of TS_3 and TS_4 would become $2(\Delta_T + \Delta_A)$. Under assumption (A2) these delays are very low, and so the pulse width would be too low. Here the addition of the delay δ_H in the local path, as shown in Figure 1, becomes important to prolong the output glitch to an uncritical length: The delayed arrival of the transition makes the local pipeline remove the (already issued) pulse later and hence start the "greater" and "greater equal" comparisons related to the next transition later.

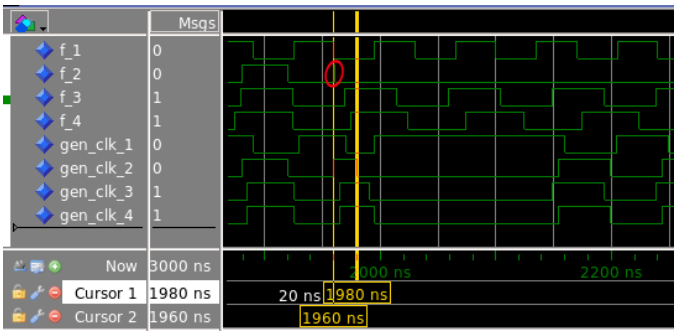

Fig. 4. Simulation trace showing a glitch of one clock source in the extended DARTS implementation

D. 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 Figure 5: On the left side of the trace the so far second fastest crystal, LRO_2, speeds up and its associated node TS_2 becomes the fastest node. As a consequence, TS_1 gets the role of the second fastest node and hence its timing dominates the output, starting from $t = 1645ns$ (see marker). As can be seen, there is only little effect on the generated clocks, apart from LRO_2 managing to speed up its associated clock output a bit – but no pulse gets shorter than δ_H. It can also be seen that it takes a couple of clock periods for TS_1 to take over the lead from T_2; this is because of the transitions buffered in the elastic pipelines.

In the same figure, further to the right, we have the case that LRO_2 suddenly slows down, now becoming the slowest clock. The end result is as expected: TS_3 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, LRO_2's change in speed is not immediately reflected in the clock outputs. In a first phase TS_2 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 $t = 1975ns$ to $t = 2255ns$ (see markers). Later on (when the LRO pipeline starts containing negative entries), TS_2 loses the lead to TS_3 and goes to the set of slowest nodes, together with TS_4. 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 TS_2) 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.

E. 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

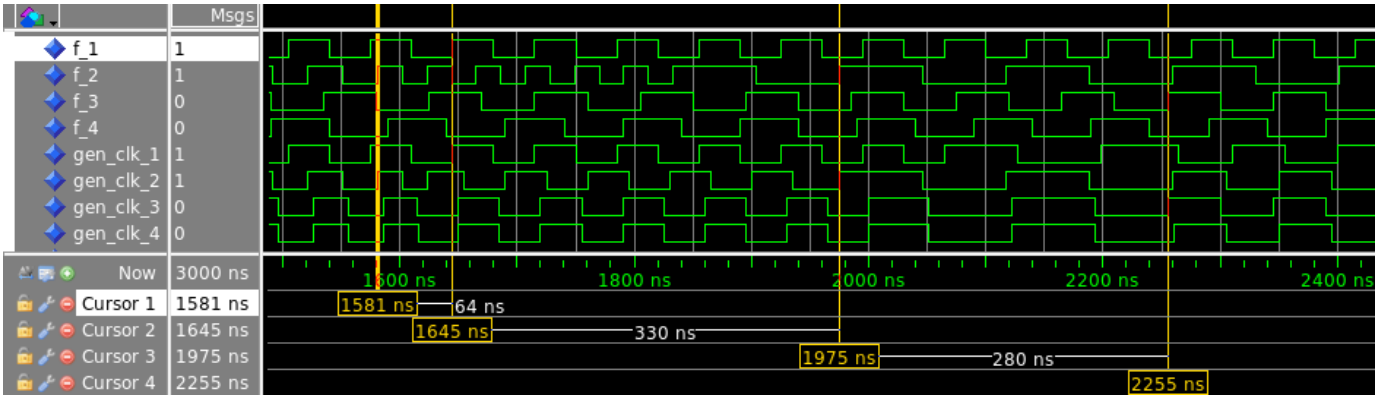

Fig. 5. Simulation trace showing the influence of a frequency change at one clock source

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_H 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 V 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.

VII. 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.

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.

References

[1] J. R. Holden, "Clock failure monitor circuit employing counter pair to indicate clock failure within two pulses," 1980, US Patent 4,374,361. [Online]. Available: http://www.google.com/patents/US4374361

[2] J. H. Hong, D. J. Shin, Y. K. Jeong, and H. J. Park, "Clock fault monitoring circuit," 1992, US Patent 5,479,420. [Online]. Available: http://www.google.com/patents/US5479420

[3] R. C. Bagley, "Method for detecting clock failure and switching to backup clock," 1996, US Patent 5,828,243. [Online]. Available: http://www.google.com/patents/US5828243

[4] W. T. Ang, H. F. Rao, C. Yu, J. Liu, I.-C. Wey, A.-Y. Wu, H. Zhao, and J. Chen, "A clock-fault tolerant architecture and circuit for reliable nanoelectronics system," in *2007 International Conference on Design Technology of Integrated Systems in Nanoscale Era*, Sep. 2007, pp. 186–191.

[5] H. Hui, Y. Changhong, C. Keming, and S. Lingling, "A novel clock-fault detection and self-recovery circuit based on time-to-voltage converter," in *2008 International Conference on Communications, Circuits and Systems*, May 2008, pp. 1204–1207.

[6] C. Yu, "A clock fault detection circuit for reliable high speed system by time-to-voltage conversion," in *2009 Second International Symposium on Electronic Commerce and Security*, vol. 2, May 2009, pp. 283–286.

[7] ——, "A novel clock-fault detection and self-recovery circuit for reliable nanoelectronics system," in *2009 International Workshop on Intelligent Systems and Applications*, May 2009, pp. 1–4.

[8] A. Gamet, Y. Bacher, S. Meillère, P. Le Fevre, and N. Froidevaux, "A simple clock-fault detection analog circuit for high-speed crystal oscillators," in *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2015, pp. 1588–1591.

[9] J. An, J. Cho, and D. Park, "On-chip glitch-free backup clock changer with noise canceller and edge detector for safety MCU clock system," in *2015 IEEE 4th Global Conference on Consumer Electronics (GCCE)*, Oct 2015, pp. 487–488.

[10] J. An, J. Youn, J. Cho, and D. Park, "Automatic on-chip glitch-free backup clock changing method for MCU clock failure protection in unsafe I/O pin noisy environment," *Journal of The Institute of Electronics and Information Engineers*, vol. 52, no. 12, pp. 2161–2170, 2015.

[11] R. Najvirt and A. Steininger, "A versatile and reliable glitch filter for clocks," in *2015 25th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Sept 2015, pp. 140–147.

[12] A. Steininger and M. Schwendinger, "A systematic approach to clock failure detection," in *2019 Austrochip Workshop on Microelectronics (Austrochip)*, Oct 2019, pp. 35–42.

[13] J. L. Welch and N. Lynch, "A new fault-tolerant algorithm for clock synchronization," *Information and Computation*, vol. 77, no. 1, pp. 1 – 36, 1988. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0890540188900430

[14] M. S. Maza and M. L. Aranda, "Interconnected rings and oscillators as gigahertz clock distribution nets," in *GLSVLSI '03: Proceedings of the 13th ACM Great Lakes symposium on VLSI*. ACM Press, 2003, pp. 41–44.

[15] S. Fairbanks, "Method and apparatus for a distributed clock generator," 2004, US patent no. US2004108876. [Online]. Available: http://v3.espacenet.com/textdoc?DB=EPODOC\&IDX=US2004108876

[16] G. Fuchs and A. Steininger, "VLSI implementation of a distributed algorithm for fault-tolerant clock generation," *Journal of Electrical and Computer Engineering*, vol. 2011, 2011. [Online]. Available: https://www.hindawi.com/journals/jece/2011/936712/

[17] W. Duer and A. Steininger, "Merging redundant crystal oscillators into a fault-tolerant clock," in *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2020, pp. 1–6.

[18] T. K. Srikanth and S. Toueg, "Optimal clock synchronization," *Journal of the ACM*, vol. 34, no. 3, pp. 626–645, Apr. 1987.

[19] I. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, June 1989.

[20] M. Függer, U. Schmid, G. Fuchs, and G. Kempf, "Fault-tolerant distributed clock generation in VLSI Systems-on-Chip," in *Sixth European Dependable Computing Conference (EDCC)*, 2006, pp. 87–96. [Online]. Available: https://ieeexplore.ieee.org/document/4020838

[21] M. Függer and U. Schmid, "Reconciling fault-tolerant distributed computing and systems-on-chip," *Distributed Computing*, vol. 24, no. 6, pp. 323–355, 2012.