The Use of Fault-tolerant Clock Synchronization Algorithms for Time Scales

Attila Kinali,*
*Max Planck Institute for Informatics, Saarland Informatics Campus, Germany

***Abstract*—Clock ensembles are at the core of many applications in which precise time or frequency is required. The widely used time scale algorithms need cumbersome modifications when clocks can be potentially faulty. Fault-tolerant clock synchronization algorithms from distributed systems allow to build time scales without the need of special detection systems for anomalous behavior of the clocks. We present a short overview of the general principles underlying these algorithms and their advantages and disadvantages when being used for time scales.**

I. Introduction

A. Background

Conventional time scale algorithms compute a weighted average of the contributing clocks (see e.g. [1]). This averaging causes problems with continuity of the time scale when a clock is inserted or removed from the ensemble.

Detection of anomalous behavior is not supported by most time scale algorithms and has to be handled by an external wrapper, often requiring additional information on the internal state of the contributing clocks to be reliable. This not only makes the system quite complex, but also incurs a trade-off: false positives of the fault detection mechanism result in healthy clocks being excluded, while false negatives result in including faulty clocks. This becomes a problem especially if clocks show a lot of intermittent faults, which is typical for harsh environments like in space applications.

The distributed systems community has been employing clock synchronization algorithms which are fault-tolerant for some time. These algorithms do not try to detect erroneous behavior, but instead mask it, which allows having a single algorithm handling the ensemble without the need of any out-of-band information. In addition, this buys time for error detection and handling; it suffices to guarantee that, at no point in time, the number of faulty clocks exceeds the limit that can be sustained. The range of faults that the system can encounter is also defined much more broadly than is usually the case with time scale algorithms. Usually, very few assumptions are made about possible faults, allowing arbitrary faults (so-called Byzantine faults). One example of these algorithms is the Welch-Lynch algorithm [2]. This simple algorithm can handle any number of faults that is smaller than one third of the contributing clocks. Here, a fault can be *any* anomalous behavior of a clock, its associated computational logic, the communication links for incoming or outgoing signals, or any combination thereof. As long as the fault threshold is not reached, the output quality degrades only slightly. With accurate clocks, the dominant source of error is the uncertainty in the measurements of the relative phases between the clocks.

B. Our Contribution

We present an overview of how to implement a fault-tolerant distributed clock synchronization algorithm. We show how the fault-masking behavior can be used to build ensembles that are immune to phase and frequency jumps of any (sufficiently small) subset of the clocks, without the need for an explicit detection circuit.

We also give an overview of the problems arising from using these algorithms in the context of time scales, namely the deterioration of the absolute noise performance compared to traditional algorithms, and discuss possible solutions.

II. Fault Model

A. Classical Fault Models

Most electronics are designed with simple fault models that restrict the behavior of the faulty nodes. Quite a few of those can be summarized in the fail-stop model, i.e. the device either works correctly and it's output is valid or the device is faulty and its output invalid. In this model it is easy to detect faults and thus also easy to deal with them in globally consistent way.

The more interesting case is deterioration of the output, like phase or frequency jumps. If these jumps are small, then detecting them becomes hard. One big disadvantage of this model is, that it only models one specific type of event (jump in phase or frequency) for only one specific type of devices (clocks). It is not possible to use results derived from these models in a broader context.

B. Byzantine Fault Model

The fault model we use here, is a very broad and strong one: In a network of n nodes, at most f faulty nodes are allowed. Unlike in most electrical and electronic fault models, these faulty nodes are allowed to show inconsistent behavior to different non-faulty nodes. Additionally, the faulty nodes are granted the power to know the states of all nodes and may communicate with all other faulty nodes using a side channel with infinite capacity. Informally speaking, faulty nodes are omniscient beings that are allowed allowed to lie.

This fault model allows to capture a wide variety of faults, even those induced by outside disturbance. Unlike in the fail-stop model, it is not possible to reliably detect which nodes are faulty in the Byzantine fault model, without incurring a huge cost in either communication or convergence time. I.e. they can mimic a correctly working node to each non-faulty node and the only way to detect them is to communicate between all nodes and identify all faulty nodes at the same time. Mickens gives in [3] an humorous view on the difficulty of dealing with the Byzantine fault model.

Even though the fault model is very strong, it offers the possibility to capture faults that are otherwise hard to model. Any system that can deal with Byzantine faults is likely to be able to deal with any kind of fault that it encounters in reality. But one must be careful about this strength as well: Algorithms that work perfectly fine in reality might fail under these model assumptions. Hence it is important to show that useful result can still been achieved with such a strong model.

III. Fault Tolerant Algorithms Under Byzantine Faults

We will now look at how to deal with Byzantine faults. For this we first need to define the inputs, outputs and the required behavior of the system. As we are interested in synchronizing clocks, we are letting each node sending a pulse in regular intervals. Each node receives the pulses from all other nodes and corrects its own pulse timing such, that eventually all (correctly working) nodes agree on a common timing and pulse synchronously.

A. General Way of Dealing with Byzantine Faults

As it is not reliably possible to detect a faulty node, another approach must be chosen to work under such strict fault models. The general approach is not to try to detect them, but instead mask potentially faulty nodes in a way, that the faulty nodes that are not masked do not cause any trouble. After masking, the remaining nodes should be used in some way that will make all correctly working nodes either agree eventually or correct themselves in a way that converges to a common value.

This shifts the problem to which nodes and how many of them should be masked. A classic result from distributed computing is, that for Byzantine agreement requires that no more than one third of the nodes are faulty (i.e. $3f < n$). In this case, one can mask the the f nodes at the top and bottom extremes and work with the $n - 2f$ remaining nodes. The intuition here is, if any faulty nodes are left within the $n-2f$ nodes, then there is at least one correctly working node with a more extreme value, thus the faulty node left will not cause a degradation. Because, in the Byzantine fault model, faulty nodes are allowed to lie, correctly working nodes might have a different view and thus select a different group of $n - 2f$ nodes. It needs to be shown separately that these different views still allow a convergence of the algorithm.

The general implementation of a single node in a fault-tolerant clock synchronization system is depicted in Figure 1. Each node collects the output signals from all nodes, including its own. The timing of the signals are measured using either a TDC or a phase comparison system. The measured timing is then fed to the algorithm, which then controls the underlying clock and shifts its phase or frequency accordingly.

B. The Welch-Lynch Algorithm

Welch and Lynch described a simple, yet effective, Byzantine fault-tolerant clock synchronization algorithm in [2]. Informally, each node sends a pulse at regular intervals to every other node, including itself. The nodes receive the pulses, sort them by arrival time and drop the f earliest and f latest pulse arrival-times. The time span of the remaining $n - 2f$ pulse arrival-times is used to calculate where the nodes own pulse "should have been." The time difference between center of the time-span (i.e. the mean of the minimum and maximum of the time-span) the nodes own pulse arrival-time is used as correction value for the next round (see Figure 2). This algorithm, despite its simplicity, is Byzantine fault-tolerant. Intuitively, no matter what the faulty nodes do, the calculated center point calculated by each node, will be within the time span of the $n - 2f$ nodes. Thus, in the absence of noise, because the each correctly working node moves into the center of the span it sees, it will, in best case, halve the distance to the other correctly working nodes. Hence the nodes will converge and pulse together. A formal proof can be found in [4]. In a real implementation, the achievable synchronization is limited by uncertainty in the measurement of the arrival of the pulses (wire delay uncertainty and noise in the measurement) and difference in frequencies of the clock oscillators. The remaining skew between clocks is bounded by

$$\max \Delta t = 2\delta + \frac{\Delta f}{f}\tau$$

, where δ is the uncertainty in the pulse arrival time and Δf is the maximum difference in frequency, which is normalized over the nominal frequency f_0 and multiplied by the round length τ. If there are no faulty nodes, the skew reduces to [4]

$$\max \Delta t_{faultfree} = \delta + \frac{\Delta f}{f}\tau$$

.

IV. The Welch-Lynch Algorithm Applied to Atomic Clocks

A. Stability vs Synchroninity

The Welch-Lynch algorithm achieves a high level of synchronization, only limited by the precision of the relative phase measurement. Given that it is possible to compare local clocks with sub-picosecond precision, this should not pose a limitation in practice. But the Welch-Lynch algorithm applies a correction at each step. This correction is derived from noisy input, i.e. it is a random variable. Thus the correction itself is a random variable and hence the Welch-Lynch algorithm acts like an integrator of a random variable. Obviously, this will lead to degradation of the stability and a conversion from white phase noise to white frequency noise. To analyze this, we have simulated a 4 node/clock ensemble with no faulty nodes. In order to make the behavior of the algorithm under different noise process more clear, we performed different simulations

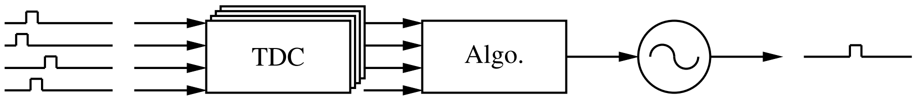


Figure 1. Design of a single node implementing a fault-tolerant clock synchronization algorithm. A TDC measures the timing difference between the different nodes. These values are fed to the algorithm which then controls underlying clock and shifts its phase or frequency. The output of the clock is distributed to all nodes.

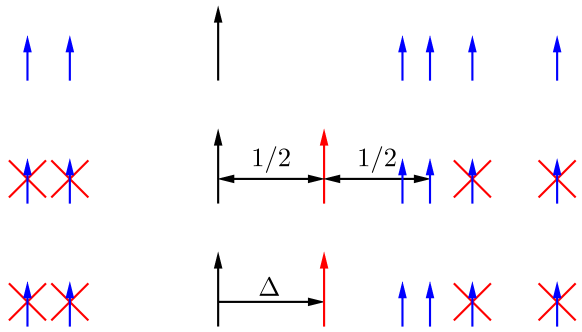


Figure 2. The Welch-Lynch clocksync algorithm. First receive all clock pulses from all nodes, including own pulse (long black arrow). Drop first $f = \lceil n/3 \rceil$ and last f received pulses (masking step). From the remaining pulses, calculate the center of the spanning time frame (long red arrow) which denotes where the pulse "should have been". Lastly, calculate the correction to be applied before the next round starts.

with only one single noise process. All 4 clocks have the same noise spectrum, but are otherwise independent. The algorithms period was chosen to be 1 s and phase corrections were applied instantaneously, i.e. before the next round starts. Measurement noise and uncertainty was set to zero, in order not to make analysis more complicated.

As can be seen in the modified Allan deviation plot Figure 4(a), the integration of the clock input noise leads to a noise process that is white frequency modulation, in the long term. Short term, below a τ of 3 s, the noise follows that of white phase modulation, though. A similar observation can be made for flicker phase modulation (Figure 4(b)), but the slope increase is smaller than for white phase modulation. This can be easily explained by the Welch-Lynch algorithm removing the outliers which get more pronounced as the low frequency components of the noise gets stronger. From white frequency modulation onwards (Figures 4(c), 4(d) and 4(e)) the slope does not degrade any further and there is only a slight degradation in stability.

Looking at the plots, one can see that there is a slight increase in slope between τ's between 1 s and about 3 s. The algorithm seems to be degraded by high frequency noise, which then ends up being integrated through the continuous corrections. Using this insight, we applied a first order IIR low-pass filter with a corner frequency of $1/100$ Hz corner frequency, at every input of each node. I.e. the phase measurements at each node are low pass filtered individually to dampen the high frequency components (Figure 3). As can be seen in Figure 4 ("LP100+WL" plots), the effect is quite tremendous improvement in stability of the ensemble. For white phase modulation, the low-pass filter brings the stability to what the underlying clocks have. For τ's longer than the filter bandwidth, the stability seems to even improve beyond what the clocks have themselves, but not much. This effect gets more pronounced for the higher exponent noise processes. While at τ's beyond the filter bandwidth, the ensemble slope is the same as the underlying clocks, at smaller τ the slope is much steeper and thus the stability improves quite considerably. Please note, that this behavior is different than a simple low pass filtered clock, where the maximum improvement is at the smallest τ (highest frequency), which the decreases until it reaches the clocks performance at the filters corner frequency. While here we have no stability improvement at lowest τ, but increasing improvement until the corner frequency is reached.

References

[1] J. Levine, "Invited review article: The statistical modeling of atomic clocks and the design of time scales," *Review of Scientific Instruments*, vol. 83, no. 2, p. 021101, 2012. [Online]. Available: http://dx.doi.org/10.1063/1.3681448
[2] J. L. Welch and N. A. Lynch, "A New Fault-Tolerant Algorithm for Clock Synchronization," *Information and Computation*, vol. 77, no. 1, pp. 1–36, 1988.
[3] J. Mickens, "The Saddest Moment," *login*, May 2013.
[4] P. Khanchandani and C. Lenzen, "Self-stabilizing byzantine clock synchronization with optimal precision," *CoRR*, vol. abs/1609.09281, 2016. [Online]. Available: http://arxiv.org/abs/1609.09281

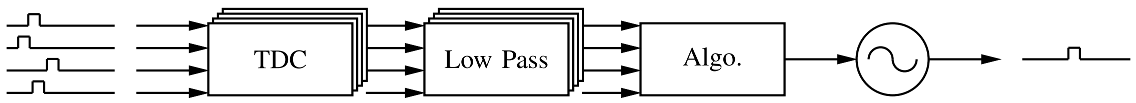
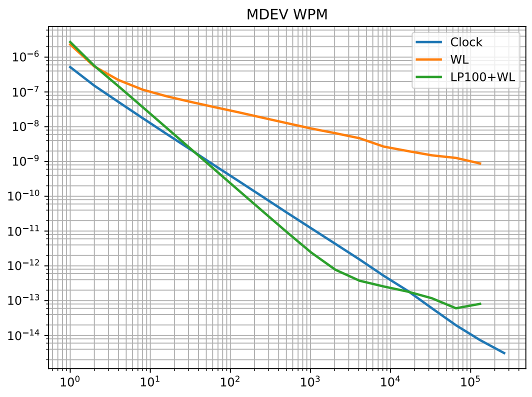
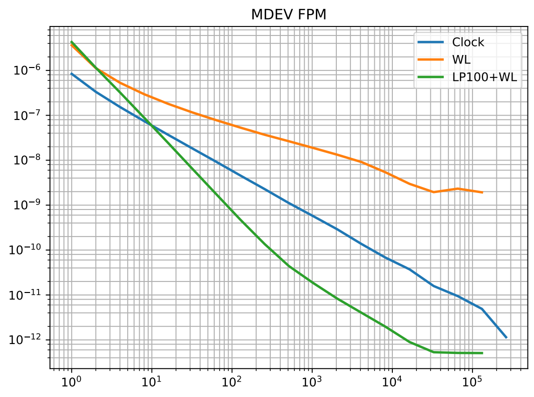


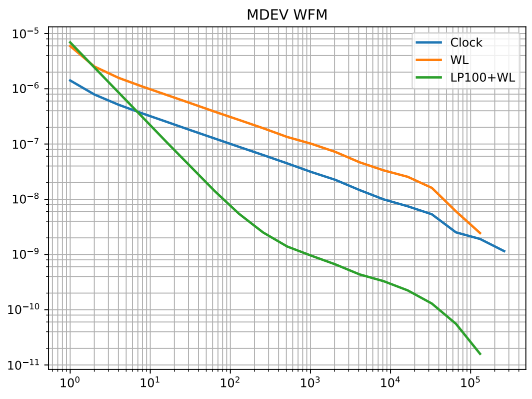
Figure 3. Additional low-pass filters between the TDC and the algorithm improves the performance of the system.



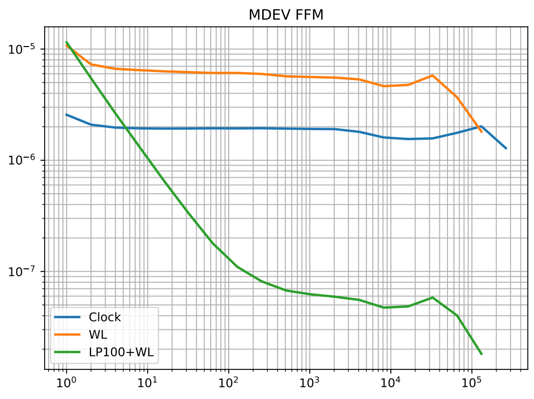
(a) White phase modulation.



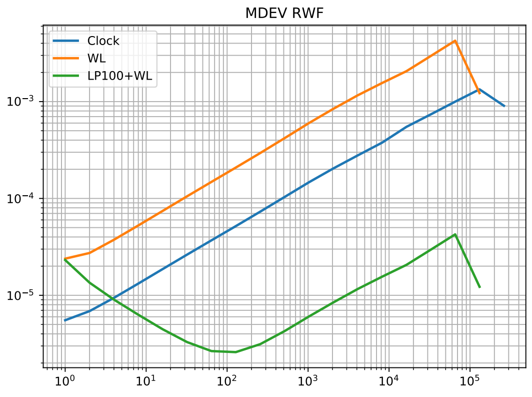
(b) Flicker phase modulation.



(c) White frequency modulation.



(d) Flicker frequency modulation.



(e) Random walk frequency.

Figure 4. Modified Allen deviation plots of a 4 node ensemble with different clock input noise spectra, but without any faulty clocks. The original clock noise (”Clock”) gets slightly degraded by the Welch-Lynch algorithm. Applying a first order IIR filter with a corner frequency of $1/100\,\mathrm{Hz}$ to each nodes input (”LP100+WL) improves the stability of the time-scale of the ensemble quite considerably, especially for the higher exponent noise spectra.