Verification of clock synchronization algorithm

(Original Welch-Lynch algorithm and adaptation to TTA)

Christian Mueller

November 25, 2005

Contents

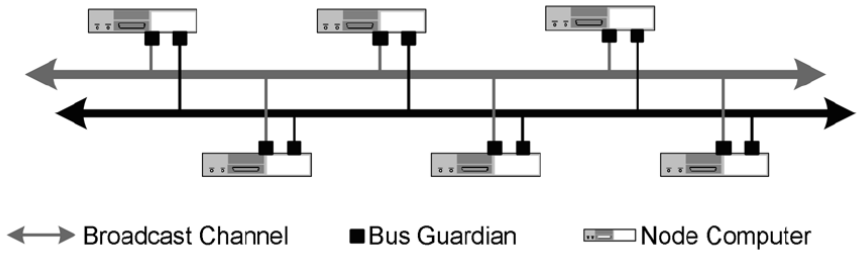
1 Clock synchronization in general

1.1 Introduction

Distributed time-triggered systems are an enabling technology for a huge set of critical technical applications, e.g. in automotive, aerospace, railways and other transportation systems, industrial automation and process control, medical systems and the like, where hard real-time requirements have to be met in a dependable, predictable manner, because people's life may depend on the services provided by this critical systems, subsystems and components.

1.2 Time-triggered and fault-tolerant

The predestination of the time-triggered cluster systems is to make some parallel work synchronous, according to the global schedules. "Time-triggered" means, that all the actions performed by the cluster components will be triggered by its internal clock at a certain local time. The basic building block of such system is a node. A node comprises a processor with memory, an input-output subsystem, a time-triggered communication controller, an operating system, and the relevant application system. At the next figure is an example for such cluster system - TTA:



The two broadcast channels connect the nodes thus forming cluster. The bus guardians prevent from sending of messages by clocks, that are out of turn.

Since we speak about critical technical applications, which could be responsible for people's life, they must be as most stable as possible. So we have to design such systems so, that they can also works, when a certain number of clusters is faulty. So it is obvious, that the clock synchronization is one of the most important problems of the time-triggered systems.

1.3 Typical problems

Each cluster component of a time-triggered system is supplied with a physical clock that is typically implemented by a discrete counter. The counter is incremented periodically, triggered by a crystal oscillator. As these oscillators do not resonate with a perfectly constant frequency, the clocks drift apart from real time. So the task of clock synchronization algorithms is the permanent computation of adjustment of a node's physical clock in order to keep it in agreement with the other nodes' clocks. The adjusted physical clock and the computed adjustment is what is used by a node during various operations and it is commonly called a node's local clock.

The general way clock synchronization algorithms operate is to make estimates of the readings of other nodes' clocks to compute an adjustment for the local clock. Since every node knows beforehand at which time certain messages will be sent, the difference between the time a message is expected to be received by a node and the actual arrival time can be used to calculate the deviation between the sender's and the receiver's clock.

One of the most difficult problems of such method are "dual faced clocks". That are faulty clocks, which "shows" different local time to different nodes. Also taking into account the clocks' drift from real time and varying message delivery times makes the problem more realistic and more challenging.

1.4 Generalised view of clock synchronization

Each clock synchronization algorithm of the described cluster systems can be presented as following:

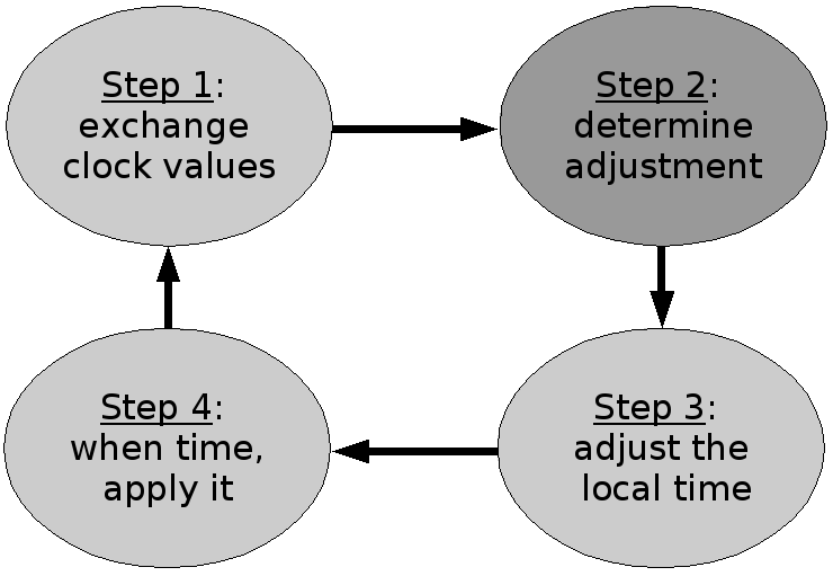
1. In the first step we read the local clock values from all local nodes in the network and store them in an array.

2. The second step is most important, because all existing algorithms differ in exactly this step. Here we compute the *adjustment*, according to the used algorithm usign the array with the stored clock readings from the step 1.

3. In the third step we compute the new local time using the correction from the step 2.

4. After waiting some neccessary time to ensure that all non-faulty nodes could compute the new clock values as well, apply the corrected local time.

The algorithms works as an endless loop in each node:



1.5 Requirements and expectations

To ensure, that a synchronisation algorithm works correctly, some properties of this algorithm should be proved. In our case there are two requirements:

1. *Agreement:* At each real time t the clock value C of each two non-faulty clocks i and j must be approximately equal, and the maximal drift must be bounded by some constant γ:

$$|C_i(t) - C_j(t)| \leq \gamma$$

2. *Validity:* the clocks of non-faulty processes must be within a linear envelope of real-time:

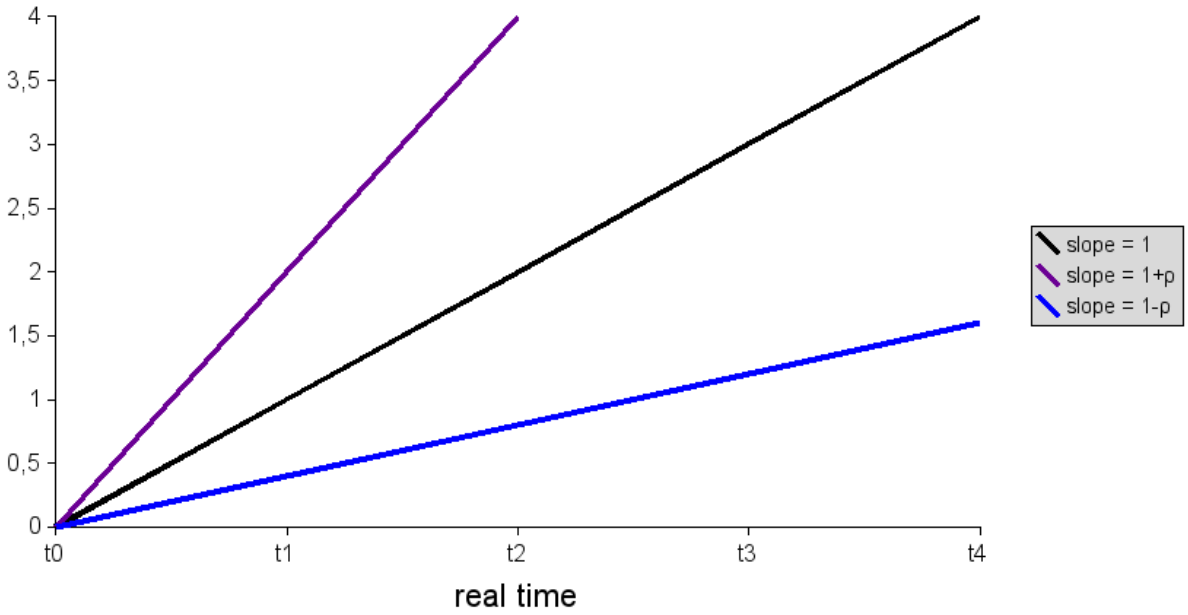
$$1 - \rho \leq \frac{d(C(t))}{dt} \leq 1 + \rho$$

The *Agreement* property ensures, that the biggest difference between each two clocks at each time is maximum γ. The *Validity* property ensures, that the slope of the increasing of the local clock values has a lower and upper bound.

Note, the function C_i is mapping the real time to the local time of the node i:

$$C_i : time \longrightarrow Localtime.$$

At the next picture we can see an example for the linear time envelope:



As we can see, the black line with slope 1 represents the idealized local time increasing. Such local clock would increment its local time value after exactly one real time unit. The violet line represents the upper bound for the increasing slope, the blue line respectively the lower bound.

2 Original Welch-Lynch algorithm

2.1 Comparison with other algorithms

In the Welch-Lynch algorithm (further: WLA) developers' model, is assumed, that all processors have access to its local read-only physical clocks, which are bounded to a very small rate of drift. Additionally each process has local time, which is obtained by adding the value of the physical clock to the value of a local correction variable. We also assume that processes are totally connected for communication. They communicate by messages, over a reliable transmission medium. There are upper and lower bounds on the length of time that any message takes to arrive at its destination. WLA runs in rounds, resynchronizing after certain time interval to correct the clocks drifting out of synchrony, and using a fault-tolerant averaging function. The size of the adjustment made to a clock at each round is independent of the number of faulty processes. At each round, n^2 messages are required, where n is the total number of processes. The closeness of synchronization achieved depends only on the initial closeness of synchronization, the message delivery time and its uncertainty, and the drift rate. There are explicit bounds on how the difference between the clock values and real time grows[1].

There are other clock synchronization algorithms that run in rounds, e.g. from Lamport and Melliar-Smith [2], from Halpern, Simons and Strong [1] and from Marzullo [3]. The three algorithms of Lamport and Melliar-Smith require like WLA a reliable, completely connected communication network and handle arbitrary faults. However, the closeness of synchronization achieved by one depends on the number of processes and that achieved by the other two depends on the number of faulty processes. In two of them, the size of the adjustment also depends on the number of faulty processes and the number of messages is exponential. Although one algorithm only needs a majority of the processes to be non-faulty, it assumes unforgeable digital signatures, The algorithm of Halpern, Simons and Strong is resilient to any number of faults (as the network remains connected), has n^2 messages complexity per round, and achieves a closeness of synchronization very similar to WLA. But the size of the adjustment depends on the number of processes and unforgeable digital signatures are necessary.

[1] [4] "A New Fault-Tolerant Algorithm for Clock Synchronization", Jennifer Lundelius and Nancy Lynch.

2.2 Assumptions

WLA can only work, if the following assumptions hold:

- The local (logical) clock value of a node will be computed as follows: $C_i = H_i(t) + R_i(t)$, where R_i denotes the adjustment computed in the last synchronization round and H_i the hardware clock of a node i i.
- The drift of the hardware clock of all nodes is bounded by some constant ρ: $1 - \rho \leq \frac{d(H(t))}{dt} \leq 1 + \rho$.
- The number n of processes is known in the whole system and each process can send messages to all other processes.
- For the number f of faulty clocks there is following constraint: $n \geq 3f$
- All clocks are synchronized in the beginning: $|C_i(t_0) - C_j(t_0)| \leq \frac{\gamma}{2}$
- The message delivery delay is limited. A message will be delivered within the following time interval: $[\delta - \varepsilon, \delta + \varepsilon]$, where $\delta > \varepsilon \geq 0$

2.3 How does it work?

We already have seen the generalized structure of all clock synchronization algorithms. The only difference is the second step – computation of the adjustment. So, how does WLA compute it? Very simply. In the first step we have send our local time to other nodes and have received and stored their local clock values in an array. Let n be the number of all nodes and f the number of all faulty nodes with $n \geq 3f$. Now:

1. sort the stored clocks $C_1, \ldots, C_n$ from smallest to largest.
2. exclude f smallest and f largest clocks from the array.
3. compute the average of the $f + 1$'st and $n - f$'th clocks.

All this steps executes the so called convergence function, the "heart" of each clock synchronization algorithm. So, the corresponding convergence function of WLA is defined as:

$$cfn([C_1, \ldots, C_n]) = \frac{C_{f+1} + C_{n-f}}{2}$$

2.4 Formal presentation

The simplified version of the Welch-Lynch algorithm for some node p looks as follows:

$T := T0$;
repeat forever
wait until $C_p = T$;
broadcast SYNC;
wait for Δ time units;
$ADJ_p := T + \delta - cfn(ARR_p)$;
$CORR_p := CORR_p + ADJ_p$;
$T := T + P$;
end of loop

on reception of SYNC message from q do $ARR_p[q] := C_p$.

2.4.1 Explanation of notation:

- The variable T always contains the local time.
- T_0 contains the initial time, e.g. 0.
- C_p is the function, which always returns the current local time.
- ADJ_p is the adjustment, computed in the current synchronization interval.
- ARR_p is the array of the node p, containing all the clock readings of other nodes.
- $cfn()$ is the convergence function.
- $CORR_p$ is the new correction (which will be added to the hardware clock)

The processes exchange at some fixed local time $C(T_0)$, $C(T_0 + \Delta)$, $C(T_0 + 2\Delta)$,...etc. their clock values by broadcasting of the SYNC message. If the local clock of some process reaches the time $C(T_0 + m\Delta)$, the local clock of any other non-faulty process will reach this local time within the interval $[C(T_0 + m\Delta) - \gamma, C(T_0 + m\Delta) + \gamma]$, since the clock drift is bounded by γ. But with the consideration of the message delivery delay, a message achieves some node in the worst case after $\gamma + \delta + \varepsilon$. And with the consideration of the bounding of the local time increasing slope we receive $(1 + \rho)(\gamma + \delta + \varepsilon)$.

So we have to wait for Δ time units to ensure, that every non-faulty node has reached the current synchronization interval, has sent its SYNC message and has received our SYNC message. Then we compute the clock adjustment, new correction using the array of clock readings from the previous round and set T to to $T + P$, where P is the duration time of a synchronization interval. At the same time we are permanent waiting for the SYNC messages of other nodes, and after receiving, we set the corresponding array cell to the current local time. $C(T_0 + m\Delta) - C(t)$ returns the time difference between my local time and the local time of the sending node.

2.5 Bounding of important constants

For a correct execution of the algorithm, P and Δ have to satisfy several conditions.

2.5.1 Bounding of Δ

The last SYNC message in the current round can arrive the node p at the time t with:

$$t \leq t_m + \gamma + \delta + \varepsilon$$

where: t_m is the real time when the synchronization round m starts, γ is the maximal local time difference between two nodes and $\delta + \varepsilon$ is the maximal message delivery delay (in order to explain the main idea, we have to assume at this point the correctness of clocks, bounded by γ, without any proof). But Δ is a local time interval, so we want to know what local time we reach at the real time t. Simple computation returns:

$$C(t_m + \gamma + \delta + \varepsilon) \leq T_m + (1 + \rho)(\gamma + \delta + \varepsilon)$$

$$\Rightarrow \Delta \geq (1 + \rho)(\gamma + \delta + \varepsilon)$$

2.5.2 Bounding of P

For each process not to miss the next round, $T + P$ must be larger than the new clock at the time of the correction! So:

$$P \geq \Delta + ADJ_{max}$$

But what is the maximal adjustment? Ignoring some boring and less important notation juggle we assume, that the adjustment of a process p is between two limits:

$$T + \delta - C_p(t_m + \gamma + \delta + \varepsilon) \leq ADJ_p \leq T + \delta - C_p(t_m - \gamma + \delta - \varepsilon)$$

For the lower bound we get

$$C_p(t_m + \gamma + \delta + \varepsilon) \leq C_p(t_m) + (1 + \rho)(\gamma + \delta - \varepsilon)$$

then since $C_p(t_m)$,

$$-(1 + \rho)(\gamma + \varepsilon) - \rho\gamma \leq ADJ_p$$

The upper bound depends on whether γ is smaller or larger than $\delta - \varepsilon$. If $\gamma \leq \delta - \varepsilon$, we obtain

$$C_p(t_m - \gamma + \delta - \varepsilon) - C_p(t_m) \geq (1 - \rho)(-\gamma + \delta - \varepsilon)$$

then

$$ADJ_p \leq (1 - \rho)(\gamma + \varepsilon) + \rho\gamma.$$

In the other case,

$$C_p(t_m) - C_p(t_m - \gamma + \delta - \varepsilon) \leq (1 + \rho)(\gamma - \delta + \varepsilon)$$

and

$$ADJ_p \leq (1 + \rho)(\gamma + \varepsilon) - \rho\gamma.$$

In both cases, we have maximal adjustment

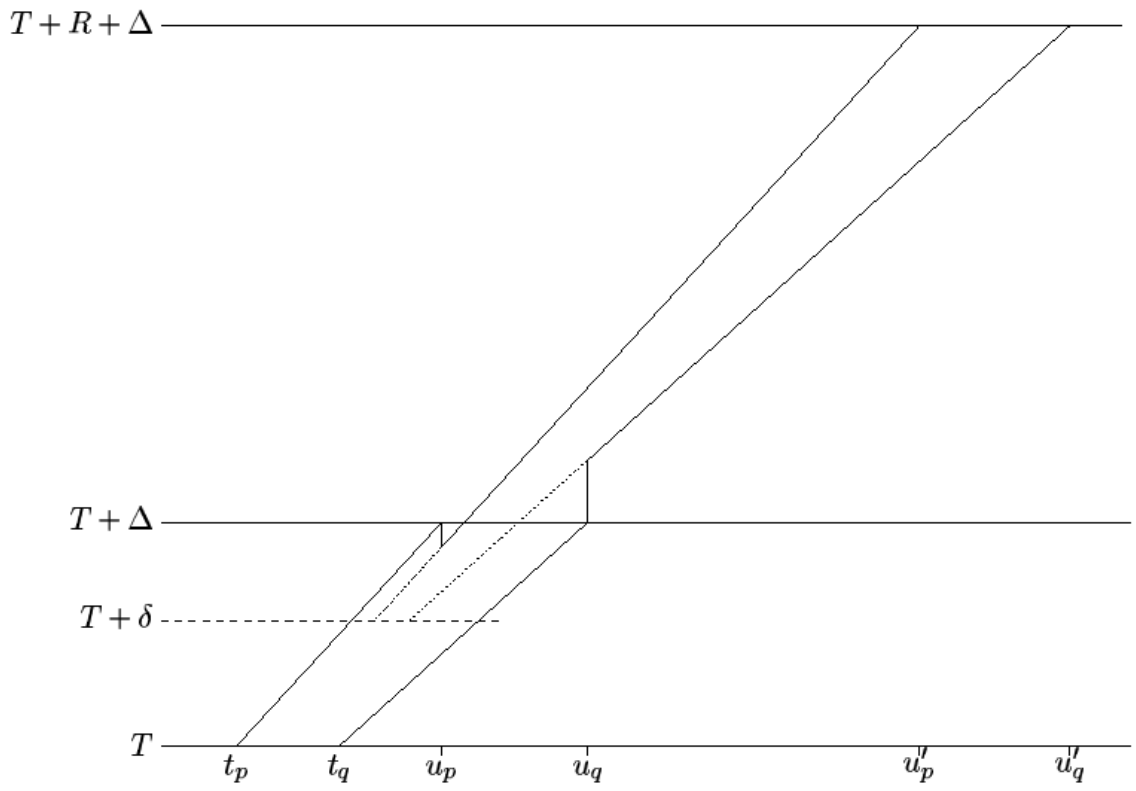
$$ADJ_p \leq (\gamma + \varepsilon) + \rho|\gamma - \delta + \epsilon|.$$

So:

$$P \geq \Delta + (\gamma + \varepsilon) + \rho|\gamma - \delta + \epsilon|$$

2.6 Pictured example from [5]

At the next picture we can see the time increasing of two clocks p and q. At the approximately local time $T+\delta$ both nodes have stored in its arrays all local clock values of all other nodes. After time Δ the correction will be applied.



Evolution of VC_p and VC_q

3 Verification

3.1 Abstract idea

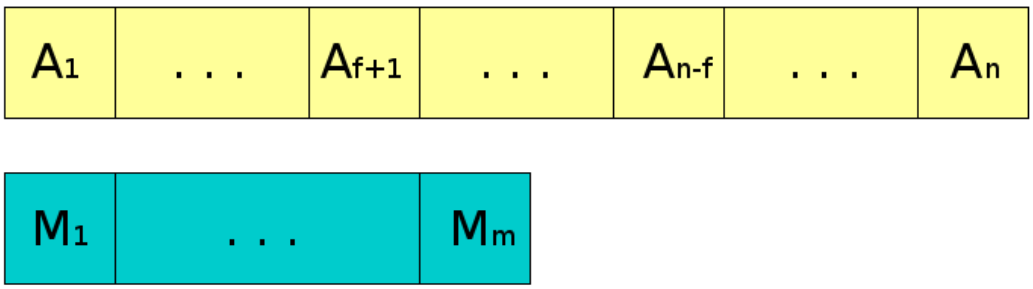
Although the algorithm is fairly simple, its analysis is surprisingly complicated and requires a long series of lemmas. To make the proof presentable, we abstract from several details and concentrate on its main idea. For simplicity we assume that broadcasting a message, computing the adjustment, storing arrival time are instantaneous operations. The idea is to examine two non-faulty clocks before a synchronization round, where the clock drift is maximal and to prove the Agreement property be induction on the synchronization round iterations. So consider the clocks of two processes p and q before the same synchronization round:

- $C_p = cfn(ARR_p)$
- $C_q = cfn(ARR_q)$

But we don't know yet, why the convergence function is a **fault tolerant** average function.

3.2 Convergence function

Let us consider the array of collected clock readings by the node p ARR_p a little bit more intently. For simplicity imagine two abstract arrays: A ($=ARR_p$) and M. M is the array, containing all non-faulty nodes from A:



What do we know about this arrays?

- They are sorted from smallest to largest.
- $M \subset A$.
- M contains all the non-faulty nodes from A and is equal at each synchronization round (but only with our simplified conditions).

- $length(M) \geq 2f + 1$ (valid by assumption $n > 3f$).

Now let us pick out some important correlations between this two arrays.

3.2.1 Proposition

$$M_1 \leq A_{f+1} \leq M_{f+1}$$
$$M_{m-f} \leq A_{n-f} \leq M_m$$

3.2.2 Proof

Proof. Since M is sorted, we have $M_1 \leq M_2 \leq \ldots \leq M_m$. By construction, $M_1, \ldots, M_m$ is then a subsequence of $A_1, \ldots, A_n$, obtained by removing fewer than f elements.

M_1 is equal to A_i for some index i. There are at least m elements among $A_1, \ldots, A_n$, which are larger than or equal to A_i, so i must be smaller than or equal to $n + 1 - m$. By the assumptions on n and m, this implies that $i \leq f + 1$ and then

$$M_1 = A_i \leq A_{f+1}.$$

Similarly, there are at least $f + 1$ elements among $A_1, \ldots, A_n$ which are smaller than or equal to M_{f+1} so

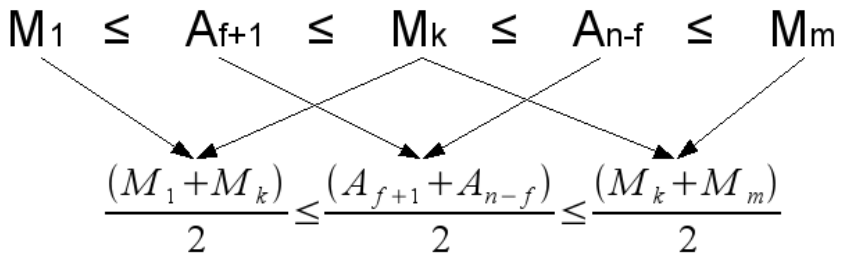
$$A_{f+1} \leq M_{f+1}$$

A symmetric reasoning proves the other part of the proposition. □

Now, let k be any index between $f+1$ and $m-f$; since $m \geq 2f+1$, such a k does exist. As a consequence of the previous proposition, we get

$$M_1 \leq A_{f+1} \leq M_k \leq A_{n-f} \leq M_m.$$

As we can see at the next figure, the average of the elements A_{f+1} and A_{n-f} is bounded only be 100% non-faulty nodes:



Now we can also bound our convergence functions of processes p and q:

- $\frac{M_1+M_{f+1}}{2} \leq cfn(ARR_p) \leq \frac{M_k+M_m}{2}$
- $\frac{M_1+M_{f+1}}{2} \leq cfn(ARR_q) \leq \frac{M_k+M_m}{2}$

As we can see, the convergence function returns a result depending only on non-faulty nodes $\Rightarrow$ **fault-tolerance**.

3.3 Proof of the Agreement property

3.3.1 Induction assumption:

$\forall p, q, t_{sync}$: p and q are non-faulty, t_{sync} is the time before one synchronization interval and the following holds:

$$|C_p(t_{sync}) - C_q(t_{sync})| \leq \gamma$$

3.3.2 Induction step: $t_{sync} \rightarrow t_{sync+1}$

Proof.

$$|C_p(t_{sync+1}) - C_q(t_{sync+1})| =$$
$$|cfn(ARR_p - cfn(ARR_q)| \leq$$
$$|\frac{M_1 + M_k}{2} - \frac{M_k + M_m}{2}| =$$
$$|\frac{M_1 + M_m}{2}| = (\gamma + P)/2$$

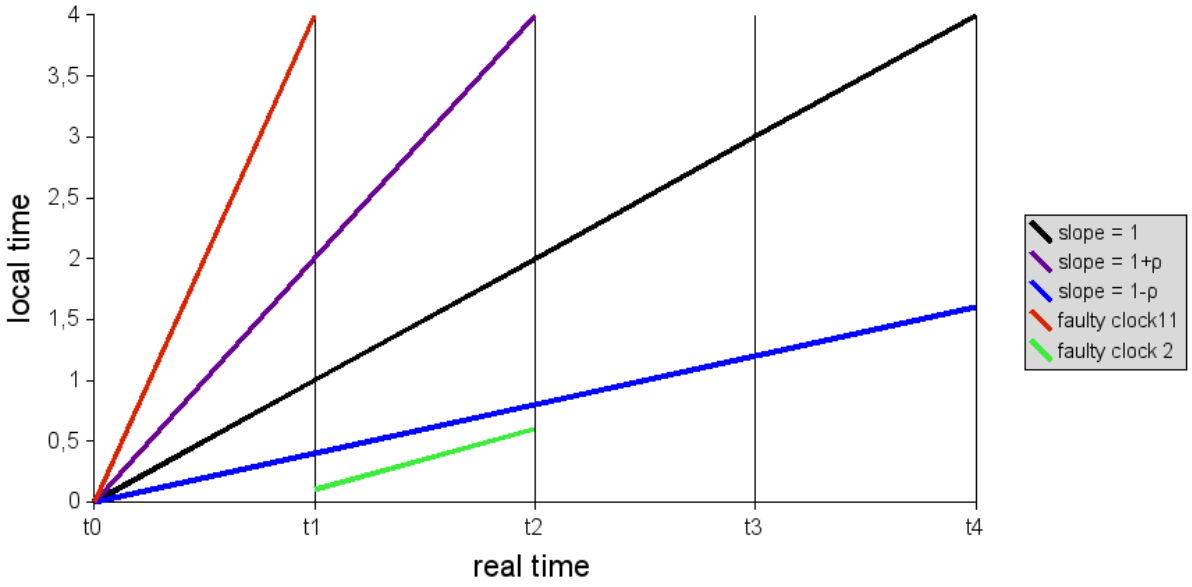
and for $\gamma \geq$ P holds:

$$(\gamma + P)/2 \leq \gamma$$

□

3.4 Proof of the Validity property

Proof. The Validity property is already particularly fulfilled by our assumption, that all hardware clocks are ρ-bounded. In general there are two cases of violation of this property. The first case, as we can see it on the next picture, is the red line:



That means, some faulty clock has a slope of the hardware time increasing, which exceeds the upper bound $1 + \rho$. But this case is impossible, since we assume, that all hardware clocks are ρ-bounded. The second case represents the green line – the slope of this time increasing is not problematic, but the correctness-value is to big, so that the clock will be shifted upward or downward. That can only happen, when the majority of good clocks always lie beyond of the correct time envelope. Can this happen? Assume we have two sets: majority set of clocks liyng in the correct envelope, but very closed to the upper or lower bound and the minority set of all other clocks within the correct envelope. After the next synchronization step, the minority clocks will compute the correctness value, which will shift them to the average time of majority clocks, but all the majority clocks compute the average time, which is a little bit shifted to the direction of the minority clocks. So the average time can never leave the correct envelope.

3.4.1 Formal:

- Since the local time function C(t) is linear, holds:

$$C(a + b) = A + C(b)$$

- Consider the local time difference of some node between two synchronization intervals:

$$C(t_{i+1}) - C(t_i) =$$
$$C(t_i + (t_{i+1} - t_i)) - C(t_i) =$$
$$T + C(t_{i+1} - t_i) - T = C(t_{i+1} - t_i)$$
$$\Rightarrow (1+\rho)(t_{i+1} - t_i) \leq T_{i+1} - T_i \leq (1-\rho)(t_{i+1} - t_i)$$

□

3.5 Disclaimer

The assumptions I have made to present the proof short and understandable are very abstract and not practical. I neglected message delivery delays and the run time of all procedures. Normally we have to bound each possible delay to a constant and then choose appropriate values for it.

4 Adaptation to TTA (Flexray)

4.1 In general

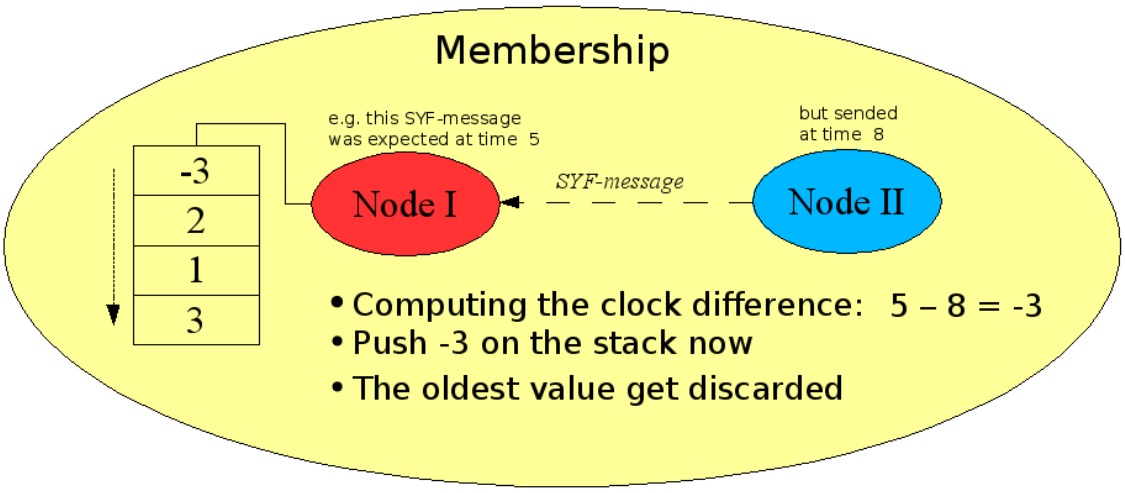
The TTA algorithm is basically the Welch-Lynch algorithm, but it tolerates only a single fault, so the WLA is specialized in this case for $k = 1$: that is, clocks are set to the average of the 2nd and $n - 1$'st clock readings (i.e., the second-smallest and second-largest). This algorithm works and tolerates a single arbitrary fault whenever $n \geq 4$. TTA exploits the fact that communication is time triggered according to a global schedule. When a node a receives a message from a node b, it computes the difference, taking into account the network delay, maximal clock drift etc.

Not all nodes in a TTA system need have accurate oscillators, because they are expensive, so TTA's algorithm is modified from Welch-Lynch to use only the clock skews from nodes marked as having accurate oscillators. Analysis and verification of this variant can be adapted straightforwardly from that of the basic algorithm. Unfortunately, TTA adds another complication. To implement the Welch-Lynch algorithm we need data structures that are independent of the number of nodes i.e., it should not be necessary for each node to store the clock difference readings for all (accurate) clocks. To determine the second-smallest clock difference reading we need just two registers (one to hold the smallest and another for the second-smallest reading seen so far), and the second-largest can be determined similarly, for a total of four registers per node. If TTA used this approach, verification of its clock synchronization algorithm would follow straightforwardly from that of Welch-Lynch. Instead, for reasons that are not described, TTA does not consider all the accurate clocks when choosing the second-smallest and second-largest, but just four of them.[2]

Because of membership algorithm TTA is able to tolerate more than a single fault by reconfiguring to exclude nodes that are detected to be faulty. So the four clocks considered for synchronization are chosen from the members of the current membership. Group membership have the property that all non-faulty nodes have the same members at all times. Next, each node maintains a push-down stack of four clock readings; whenever a message is received from a node that is in the current membership and that has the SYF field set, the clock difference reading is pushed on to the receiving node's stack (ejecting the oldest reading in the stack). Finally, when the current slot has

[2] [11] "An Overview of Formal Verification For the Time-Triggered Architecture", John Rushby.

the synchronization field (CS) set in the MEDL (Message Descriptor List), each node runs the synchronization algorithm using the four clock readings stored in its stack.



Formal verification of the TTA algorithm requires more than simply verifying a four-clocks version of the basic Welch-Lynch algorithm: for example, the chosen clocks can change from one round to the next. However, verification of the basic algorithm provides a foundation for the TTA case.

4.2 Fault assumptions

In TTA bus topology and in a Flexray system there is no dual faced clock effects possible, since each node uses the bus, to send a message, so all nodes receive all messages at the same time. Do we still need the Welch-Lynch algorithm? Yes, because some messages can get lost because of sender and receiver faults. For example, if some node has expected four messages, but received only three of them, the fourth clock value is set to null and will be filtered out as a faulty clock by the convergence function of Welch-Lynch-Algorithm.

4.3 Further changes

In Flexray and TTA each node starts a synchronization round at different time, so the duration of one round P have to be changed accordingly to this.

5 Conclusion

The Lynch-Welch algorithm is very effective, fault-tolerant clock synchronization algorithm for time-triggered systems, which requires relatively small network traffic and is simply to implement. The proof of this algorithm is surprisingly complicated because of mutual dependencies of many assumptions. Also many important delays must be bounded, depending on each other. For relatively short and understandable summary of the proof, which preserve all important lemmas etc. I would refer to [5]. [9] contains very detailed improved Shankar's proof of the abstract Schneider' algorithm and its instantiation (Welch-Lynch algorithm) by EHDM proving system.

6 Bibliography

1. J. Halpern, B. Simons and R. Strong, Fault-tolerant clock synchronization, to appear in Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing (1984).

2. L. Lamport and P. M. Melliar-Smith, Synchronizing clocks in the presence of faults, SRI International Report (March 1982).

3. K. Marzullo, Loosely-coupled distributed services: a distributed time service, Ph.D. dissertation, Stanford University (1983).

4. "A New Fault-Tolerant Algorithm for Clock Synchronization", Jennifer Lundelius and Nancy Lynch, Laboratory for Computer Science Massachusetts Institute of Technology Cambridge, MA 02139, June 1984

5. "The Welch Lynch Clock Synchronization Algorithm", Bruno Duterte, Technical Report 747, 1998.

6. "Flexray Communication system", Protocol specification, v2.1.

7. "The Time-Triggered Architecture", Hermann Kopetz, Fellow, IEEE Guenther Bauer.

8. "Mechanical Verification Of clock Synchronization Algorithms", D. Schwier and F. von Henke.

9. "Verification of Clock tolerant Verification Systems", Paul S Miner, NASA, 1993.

10. "Formal analysis of Fault-tolerant Algorithms in the time-triggered architecture", Holger Pfeifer.

11. “An Overview of Formal Verification For the Time-Triggered Architecture”, John Rushby.

12. “Understanding Protocols for Byzantine Clock Synchronization”, Fred B. Schneider, 1987.

13. “Formal Verification for Time-Triggered Clock Synchronization”, Holger Pfeifer, Detlef Schwier, Friedrich W. von Henke.

14. “Verteilte Algorithmen”, Juergen Schoenwaelder.