Time synchronization for an Ethernet-based real-time token network*

Ferdy Hanssen, Joost van den Boom, Pierre G. Jansen, Hans Scholten
Distributed and Embedded Systems group
Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente
PO-Box 217, 7500 AE, Enschede, the Netherlands, FAX: +31 53 489 4590
hanssen@cs.utwente.nl

Abstract

We present a distributed clock synchronization algorithm. It performs clock synchronization on an Ethernet-based real-time token local area network, without the use of an external clock source. It is used to enable the token schedulers in each node to agree upon a common time. Its intended use is in resource-lean systems, where heavy-weight protocols like NTP cannot be used.

We present a short overview of the working of the protocol, as well as experimental results.

1. Introduction

Running tests with a prototype of our Ethernet-based real-time token network [1] revealed that the protocol goes awry when the clocks are not synchronized. The distributed token scheduling algorithm needs a notion of global time. So we had to come up with a clock synchronization protocol which adheres to the requirements of our network.

The network, called RTnet, has the following requirements:

- based on existing hardware and protocols wherever possible; this will help to keep the costs down, a project prerequisite. E.g. Ethernet is proven technology in both protocols and hardware, and it is cheap and dependable;
- QoS guarantees for real-time traffic; we want to support both high-bandwidth multimedia traffic and low-bandwidth control traffic;
- open for non-real-time traffic;
- fault recovery; the network should be robust against failing nodes;
- plug-and-play; nodes need to be able to join and leave the network at will.

Since the network is intended for in-home use, and the goal is also to have devices with low resource capacities, such as processing power or memory, we want to use a clock synchronization protocol which does not need much processing or memory. Yet it should be accurate enough for our network protocol to function. Currently, our prototype operates with units of 10 ms, the best our hardware can provide reliably.

*This work is sponsored by the Netherlands Organisation for Scientific Research (NWO) under grant number 612.060.111, and this work is supported by the IBM Equinox programme.

2. Related work

Clock synchronization has been researched [2] since Lamport [3] wrote about it. It consists of two parts: synchronization of time and of frequency. Both need to be used together if fully synchronized clocks are to be obtained.

The best-known and most-used clock synchronization algorithm in use today is the Network Time Protocol (NTP) [4]. This protocol relies on time servers, which are categorized in so-called strata. The time servers in stratum 1 have the most exact notion of time, with time servers in stratum 2 having a less exact notion, and so on. Servers synchronize with servers in their own stratum or the stratum directly preceding theirs, that are given a higher priority. Network distances between the servers are continuously measured and taken into account when synchronization messages are exchanged.

Another algorithm being used today is the Distributed Time Service (DTS) [5], incorporated in the Distributed Computing Environment (DCE) of the Open Group [6]. Here, every LAN has some local time servers, which are used by so-called time clerks in the nodes on the LAN to synchronize their clocks. When not enough local time servers are available, a time clerk can also communicate with a global time server, using so-called courier time servers. These are used to synchronize clocks between LANs.

The main disadvantage of these algorithms is the fact that they do not really take resource usage into account. They are designed to operate on large networks that consist of several smaller networks, and thus have fairly complex communication protocols to account for that. Our network is much smaller, and therefore does not need the overhead induced by these protocols to support internetworking systems.

3. The network architecture

Our network protocol operates using a timed token on a broadcast capable network. The broadcast capability is needed for our current way of handling new nodes joining the network. An important aspect of the token is that it does not follow a predefined ring of all nodes.

We distinguish two kinds of network traffic: real-time and non-real-time. Real-time traffic has precedence over non-real-time traffic, which means that non-real-time traffic is sent only when no real-time traffic is being offered by

any node. As soon as a node has real-time traffic to send, it will obtain the token. Non-real-time traffic is organised by having the token travel from node to node using a Round Robin paradigm, with each node having the same Token-Holding Time (THT) for non-real-time traffic.

Real-time traffic is allowed in the form of real-time streams. Each stream is characterised by the bandwidth it needs and a period. A node is then granted access to the network once every period for every stream that originates from that node. When a node has access to the network (i.e. it has the token) for a particular stream, it may transmit messages for a duration of time: the real-time THT. This THT can be calculated easily using the period and the bandwidth of the stream, and the release times of all the other streams. To enable these calculations, all relevant network state, i.e. basically the list of nodes and their streams, is present in the token itself. In this way every node can make decisions concerning the entire network.

Our network supports pre-emptability of streams. This means that a node, which is transmitting real-time messages for a given stream, has to give up the token to another node with a higher-priority stream than its own. The token will return to this node automatically to enable it to finish the transmission of the messages for the current period of that stream. RTnet uses a real-time scheduling algorithm, currently Earliest Deadline First (EDF), to guide the token through the network, and an admission test to make sure the real-time streams on the network remain feasible.

Robustness is handled by using a so-called monitor node to keep track of progress of the token through the network. This monitor task is also distributed, and the monitor node is the node that held the token directly before the node currently holding the token. When it detects that the token holder does not forward the token in time, it will come into action and will regenerate a token when the token holder does not respond to its poll.

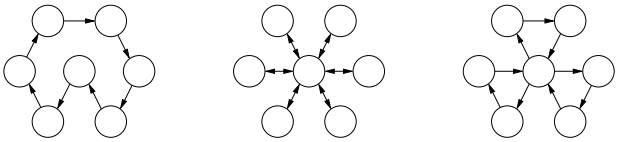
4. The time synchronization protocol

For our purposes an internal clock synchronization protocol is sufficient, as we only want the nodes participating in the protocol to agree upon a certain clock, they do not have to agree with the time of some external time server. The time synchronization protocol for this network should be simple and light-weight, we do not want more synchronization messages than absolutely necessary and we do not want to keep a history of past clock values for statistically derived improvements, in order to keep the resource usage of the synchronization protocol to a minimum. We assume that clocks do drift and that message delays are unbounded. Furthermore, we assume reliable, trusted processes, so no security mechanisms like authorization are needed, and we assume reliable messages, i.e. no messages will get lost.

Our clock synchronization protocol consists of four stages:

1) initialization;
2) message exchange and offset calculation;
3) time and frequency adjustment;
4) finalization.

In short, messages with clock values are exchanged, from which the synchronization master computes a new, common clock and distributes this to the others using a broadcast. All nodes then adjust their clocks in a continuously differentiable way to the new value.



(a) Ring. **(b) Star.** **(c) Flower.**

Figure 1. Topologies.

4.1. Initialization

A node that wants to join an RTnet network, listens for an announcement message. It will reply to this message, stating it will join, and is then added to the network by the node that sent the announcement. The new node sets its clock to the announcement message's time stamp. This is the initialization phase, the new node's clock is now approximately one network delay behind, but it will be synchronized during the next synchronization rounds.

4.2. Message exchange and offset calculation

Synchronization messages can be exchanged between nodes in a simple ring or star shaped pattern, as in figures 1(a) and 1(b). A synchronization algorithm which exchanges messages in a ring, has the advantage of needing only $n + 1$ messages for n hosts: n messages to travel the ring and 1 message to broadcast the new time. However, as n increases, so does the calculation error, originating from the fact that the propagation delay in the network is not a constant, but it is assumed to be bounded by the interval $[\delta - \epsilon, \delta + \epsilon]$, where δ is the average measured delay.

A synchronization algorithm, which exchanges messages in a star has the advantage of a small maximum error of 2ϵ. However, it needs $2n - 1$ messages to synchronize n hosts: $2(n - 1)$ time stamp messages and 1 message to broadcast the new time.

We will use a combined topology, called a flower (see figure 1(c)), which is a combination of rings of 3 nodes, always including the master node, and a possible ring of 2 nodes to include a single remaining node. This topology has an error of $(n - 1)\epsilon$, which is of the same order as the star topology, but is much smaller than the error made in the ring topology. The number of messages that needs to be exchanged in this flower topology is $3(n - 1)/2 + 1 < 1.5n$ when n is odd and $3(n - 2)/2 + 3 = 1.5n$ when n is even. This is significantly better than the star topology.

The center node in the flower is the *synchronization master*. This master is the node that happens to have the token when the *synchronization timer* expires. This master starts the synchronization round by going around the leaves to obtain the current clock values, compute the desired clock value, using a method described in the next section, and broadcast this value.

It may also be desirable to designate one node to always be synchronization master. E.g. when the burden of computing new clock values is too much for some of the nodes due to resource contraints, or when one node has access to an external clock source. This of course reduces the robustness, as an algorithm has to be devised to have another node pick up this task when the original synchronization master leaves the network. But the possibility of using an external source to also synchronize the network clock with the real world clock may be a large advantage for the applications running on the network.

4.3. Time and frequency adjustment

At the end of a round of message exchanges, a new time has to be computed, using the gathered values of all the clocks. There are several possibilities, such as averaging or taking the median, all with their own properties for error propagation or computation time. Averaging all clock values is not a good idea, because it is very prone to error when some clocks are very much off. Taking the median needs more computation, because you need to sort all clock values, but some clocks which are much off track do not influence the outcome. The median, however, does also not take into account the distribution of the clocks. Using the trimean (add the 25^{th} percentile, twice the 50^{th} percentile, and the 75^{th} percentile, divide by 4) is a bit more prone to errors than using the median, but it represents the actual clock values better.

Tests with an implementation in a prototype of RTnet have shown that using the trimean provides the best results. Figure 2 shows the maximum clock difference between three nodes during experiments using the three different methods. The network consisted of three low-end PCs running standard Linux, as the protocol should work on resource-lean systems. Each experiment was carried out twice, resulting in two lines per graph, and each experiment lasted fifteen minutes. The figure clearly shows that the difference in clocks between the nodes is smallest when using the trimean.

The large values in the first thirty to sixty seconds mean that the nodes are not synchronized yet, and the other spikes may be explained because a non-real-time operating system was used in the test. Therefore operations being carried out in other parts of the operating system may have influenced our measurements. But a maximum difference of 150 to 200 microseconds is not fatal, as the timing used in the real-time network protocol has to be accurate in the 10 ms range. Our prototype currently operates with units of 10 ms, as this is the best the current Linux kernel can do reliably on our hardware.

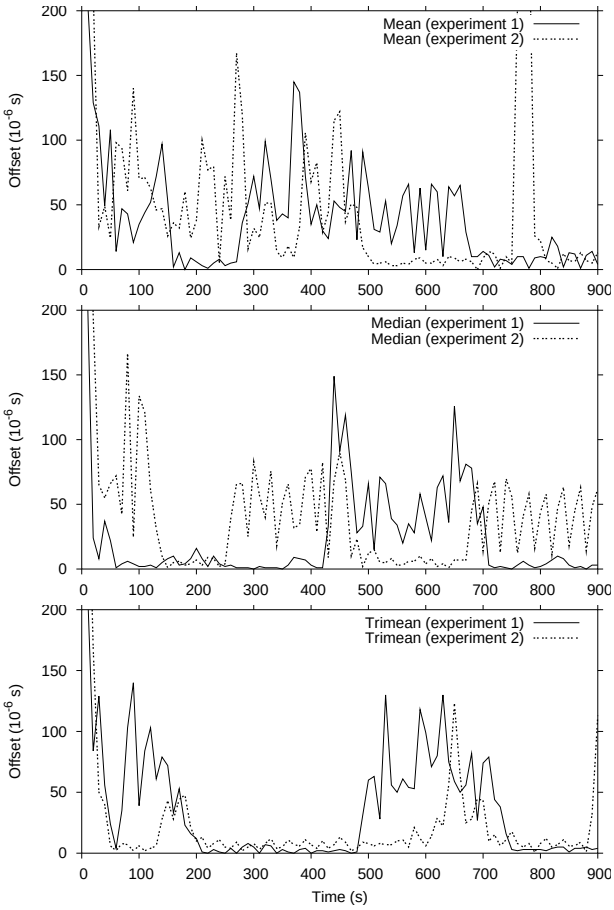


Figure 2. Comparison of time computation algorithms: mean, median, and trimean.

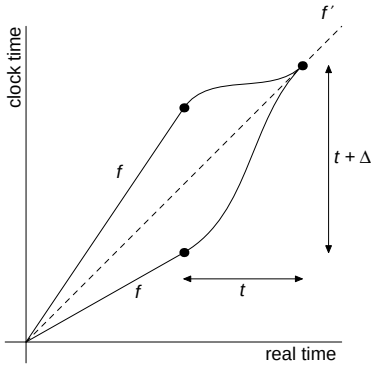


Figure 3. Continuously differentiable clock adjustment.

Each node is responsible for adapting the time and frequency of its own clock when it receives the broadcast with the new time. The node knows it should adjust its clock with Δ time units, and it knows the synchronization period T. Thus, the node can calculate the frequency f' it should have from the current frequency f using $f' = f/(1 - \Delta/T)$. For time adjustment the node knows it should advance its clock with $t + \Delta$ time units during the next t time units. The node knows everything needed to adjust the clock in a continuously differentiable way as in figure 3.

We experimented with two approaches: analytical clock adjustment and a closed-loop control system. The analytical clock adjustment is based on the fact that the third degree polynomial to express the adjustment can be derived using the two endpoints (x, y) and $(x+t, y+t+\Delta)$, and the derivatives in those two points, f and f' respectively. Using this polynomial the values of the clock and frequency for the adjustment can then be computed.

The closed-loop control system uses a set point to guide the system to the desired state. The current state of the system consists of the current offset Δ, by which the time of the clock should be adjusted, and the current frequency f of the clock. The control function for the offset used is $\dot{\Delta} = 1 - f/f_{\text{set}}$, where f_{set} is the desired frequency. The control function for the frequency used is $\dot{f} = k_1\Delta + k_2(f_{\text{set}} - f)$. The constants k_1 and k_2 should both be positive, and their exact values can be determined experimentally, as they control the speed with which the clock will settle.

The analytical and closed-loop methods have been simulated, and the results are shown in figures 4 and 5. Both algorithms have been simulated with a slow-running clock at a frequency of 0.5 and an offset of 10 time units by which the clock should be adjusted. The clock should be adjusted after 50 time units. The constants k_1 and k_2 for the closed-loop system were chosen as 0.025 and 0.3 respectively.

The closed-loop method tries to bring the offset to zero quickly (it had done 90% of the necessary adjustment in half the time needed for the entire adjustment), whereas the analytical method had done 90% of the necessary adjustment after 80% of the time needed to adjust entirely. This means that the closed-loop system can let the frequency settle gently, where the analytical method does this much more abruptly.

The analytical method completes with an offset just below zero, which can be explained by the fact that, although the method is continuous, an implementation, and thus our simulation, cannot be, as computers are not continuous but digital machines. The simulation uses a step size of 0.5 time units for the computation, introducing small errors, which can be seen in the final offset. The closed-loop system does not suffer from this problem, as it always uses the previous frequency and offset to determine the new frequency, where the analytical method does calculates the polynomial, and thus all frequencies at the beginning. Because of these results, we chose to use the closed-loop system in the prototype.

4.4. Finalization

The finalization phase consists of scheduling the next synchronization round. As synchronization needs to be done periodically, it is scheduled just like any other real-time stream, but the period of these rounds can change. If the clocks behave nicely, we do not have to synchronize as often as when the clocks behave more erratically. We calculate the maximum drift rate using the old synchronization period and the maximum offset of the node's clocks, and we then calculate a new synchronization period using this maximum drift and the maximum offset that we allow the nodes to have.

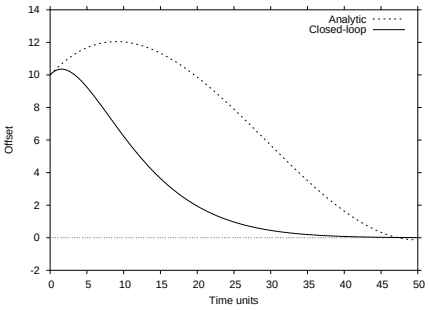


Figure 4. Offsets during adjustment.

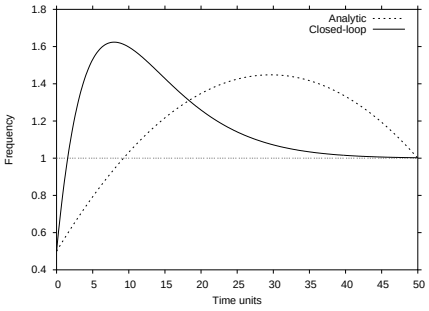


Figure 5. Frequencies during adjustment.

5. Future work

The clock synchronization algorithm is used in a prototype of RTnet and we believe it to be correct. We are currently working on a formal proof. We are also looking at the possibility of using an external source to match our network clock with the real time. Furthermore, we will investigate the influence of the operating system, especially the network drivers, on the performance of the time synchronization protocol.

References

[1] F. Hanssen, P. Hartel, T. Hattink, P. Jansen, J. Scholten, and J. Wijnberg, "A real-time Ethernet network at home," in *Proceedings Work-in-Progress session 14th Euromicro international conference on real-time systems (Research report 36/2002, Real-Time Systems Group, Vienna University of Technology)*, M. G. Harbour, Ed., Vienna, Austria, June 2002, pp. 5–8.

[2] B. Simons, J. L. Welch, and N. A. Lynch, "An overview of clock synchronization," IBM Research, Tech. Rep. RJ 6505, Aug. 1988.

[3] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[4] D. L. Mills, "Internet time synchronization: the network time protocol," *IEEE Transactions on Communications*, vol. 39, no. 10, pp. 1482–1493, Oct. 1991.

[5] *Technical Standard — DCE 1.1: Time Services*, X/Open Company Limited, 1994, X/Open Document Number: C310, ISBN 1-85912-067-9.

[6] "Distributed Computing Environment web site," http://www.opengroup.org/dce/.