Evaluating a Clock Synchronization for Dependable Sensor Networks

Spiro Trikaliotis[1], Georg Lukas[1]

[1]University of Magdeburg
Institute for Distributed Systems
Universitätsplatz 2, 39106 Magdeburg, Germany
{spiro,glukas}@ivs.cs.uni-magdeburg.de

Abstract

A synchronized clock is an important prerequisite for many distributed algorithms. This clock is used to give an "occured before" relationship, as well as for synchronizing distributed actions. There are many clock synchronization algorithms with varying precisions and assumptions on the underlying network topology. In this paper, a synchronization protocol is presented which achieves a high precision in the order of 20us to 30us in a one-hop wireless environment, and a multiple of this value for multi-hop wireless networks, such as sensor networks. The protocol works reliably even if message losses occur, which is very likely in wireless networks. For this, it utilizes redundancy in the sent time information. This protocol is implemented and evaluated on standard PC hardware running RT-Linux/Free, and an outline of the extension for multi-hop scenarios is given.

1. Introduction

The synchronization of clocks in a multi computer environment is very important for many applications. Distributed computers use their clocks to synchronize their computations, or to serialize distributed events throughout the networks. This is especially true for real-time systems and sensor-networks, as these have to interoperate with their environment. In a wireless network, the protocol has to cope with much more message losses and variances in the medium access than with wired mediums. This is a challenge especially when building dependable sensor networks.

Most existing wirelss clock synchronization protocols assume that all stations are within direct reach of each other because they exploit the broadcast property of the medium to achieve a high precision. Our protocol utilizes the broadcast property of the medium, but still allows for ordering distributed events, as necessary for sensor networks.

2. Related work

The synchronization of clocks is a rather old topic, but it has got more attention recently for wireless and sensor networks.

A good survey on clock synchronization for sensor networks can be found in [7]. It shows that traditional clock synchronization protocols for wired networks cannot be used for wireless sensor networks because these require the ability to adapt dynamically, the ability to handle sensor mobility, and scalability. The sensors themselves are heavily resource-constrained because of limited battery power. Furthermore, they need to operate in highly lossy and unreliable environments. As a result, several clock synchronization protocols for wireless sensor networks have been designed in the recent past. We presented a protocol in [5], [6], which is based on an idea which can be found in [1], [8]. The idea is also utilized in the clock synchronization protocol of the CAN standard [3].

The underlying idea is to minimize the time critical path of synchronization in order to improve the quality of the synchronization.

There has been much other work on clock synchronization. [2] showed an approach similar to [5], but focusing more on sensor-networks and their low-power constrains as well as their multi-hop nature.

For this purpose, they calculate a linear regression for the time values obtained from a time server, and they propose a scheme for synchronizing time for more

1-4244-0054-6/06/$20.00 ©2006 IEEE

than one broadcast domain. Because of their high computational requirements, they are not best suited for sensor networks with small, battery-constrained devices. [9] tries a similar approach with the same disadvantages.

3. The clock synchronization protocol

3.1. Critical path minimization

The main idea of this clock synchronization protocol is to minimize the critical path in the transmission of the clock information.

Time synchronization protocols using message delivery for communication all contain a timing critical path. The lower the jitter of this critical path, the better precision is achievable [4]. Figure 1 shows the critical path for conventional clock synchronization algorithms.

Most synchronization protocols operate in the following way: Some station takes a timestamp and sends it over the network. This way, the critical path contains local processing of the message at the sender, at the receiver, and the medium access which can be very high on CSMA (carrier sense multiple access) networks like Ethernet and most wireless networks.

Note that the station taking the timestamp differs whether there is a master/slave protocol, or a distributed one. In distributed protocols, some or even all stations take and send the timestamp. In master/slave protocols, only one master takes a timestamp. Subsequently, we will only the case of master/slave protocols for brevity.

In this case, the time consumption is as follows: Whenever a synchronization round starts, a time sender prepares a packet to be sent over the network at time instance t_1. It puts a timestamp T_{sender} into that packet. This packet is processed on the master machine. This includes operating system overhead and overhead from the medium access mechanism. At t_2, the packet leaves the sender and is send out to the communcation medium. At time t_3, the recipient physically receives the packet. Again, the network card and the operating system are involved. It takes the receiver up to t_4 to process the packet in the time synchronization algorithm.

The timestamp T_{sender} corresponds to the instance of time t_1. On processing of the packet on the receiver at t_4, the time has progressed by $t_4 - t_1$. Obviously, the jitter of $t_4 - t_1$ is crucial for the achievable accuracy of the synchronization protocol. Thus, this is the critical path of the synchronization protocol. Unfortunately, this critical path contains many parts which are very variable in their running time. The operating system has to process the packets two times, at the sender's (t_1 to t_2) and at the receiver's site (t_3 to t_4). Here, multitasking and caches of the processor can influence the time, resulting in a jitter. An even bigger jitter is generated due to the medium access time, which is part of the critical path. Even if the user can prevent most of the jitter somehow on the operating system level - by using a real time OS, clever programming, or whatever -, there is still the waiting time for the access of the communication medium in there (t_1 to t_2). This medium access time is unpredictable by nature.

Note that the delay $t_4 - t_1$ could be calculated and corrected for if the jitter where zero. Anyhow, this is not possible for the jitter, as that one is unknown.

In contrast, our protocol moves most of the operating system time and all of the medium access time out of the critical path. To achieve this, an *a posteriori* approach, involving two messages, is used to synchronize slaves to a master. The protocol uses an event which can be observed by every recipient and the sender itself. This globally observable event is a special message, the indication message. Figure 2 shows the timing and the critical path of our protocol.

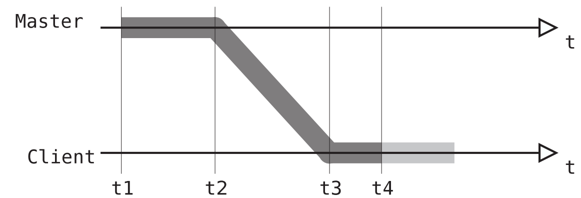

Figure 1. The critical path of other synchronization protocols

At time t_1, the master prepares the global event, the so-called indication message. The operating system gets this message, the medium contention takes place, and at t_2, the message is sent out to the medium. At time t_3, this message physically arrives at the slave. At t_4, the slave takes a timestamp T^S of this message. As the master itself knows when exactly it sent out this message, it can generate a timestamp T^M at approximately t_4, too. We will discuss the word "approximately" later on in this section. Furthermore, note that we ignore the propagation delay through the wireless medium, which is orders of magnitudes lower than the rest of the timing.

Now, the master has to tell the slave its timestamp T^M to allow the slave to adjust its clock. For this, it sends another message at t_5 which is processed at the client at t_6. Now, the receiver knows its time dif-

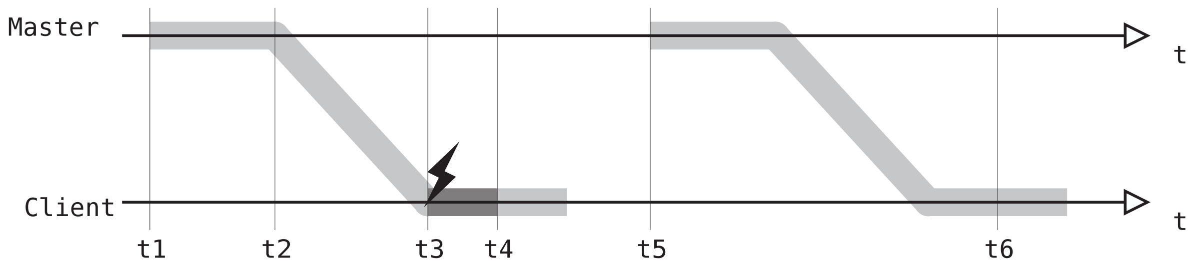

Figure 2. The critical path of the clock synchronization protocol

ference to the sender at t_3 by calculating $T^S - T^M$. As this information is not the current difference to the sender's time at t_6 anymore, the receiver's time may have drifted away from the sender's time much more. Thus, it does not make much sense to apply the difference directly to the receiver's time, but use a more sophisticated clock correction. This correction gives a continuous clock, and it is discussed in the next section.

If the routine for taking the timestamps is placed into the driver of the network interface card - the best place for this is the interrupt service routine -, the jitter of $t_4 - t_3$ can be made very small.

Obviously, the critical path has been minimized to the path t_3 to t_4, which does not contain any medium access anymore, and only a minimal amount of operating system and cache dependancy. Thus, the jitter is very low. Note that the second message sent at t_5 is not timing critical.

Instead of using two messages as states above, an easy optimization of this algorithm is to use only one message for the indication, the global observable event, in one synchronization round, and to send the timestamp of the sender for the last round in the same message. This way, only one message is needed for synchronization.

For practical purposes, often, it is not possible to determine the point of time when a message was sent exactly, that is, t'_3 on the master might be different to t_3 on the slave. Due to this, the timestamp T^M of the master might refer to another point of time t'_4 than t_4. This is due design decisions in modern network cards and their drivers to optimize throught. For example, they using buffering, interrupts and direct memory access (DMA) to act as much autonomous as possible. Unfortunately, this makes it harder to determine the exact instant of time when the message was put on the medium.

Note that this is no fundamental problem for our protocol, as this can be circumvented in different ways. For example, network cards can change their operational mode to not use buffering. Unfortunately, this would reduce network throughput, and it might increase the processing load on the host computer. If buffering can be disabled on a packet basis, the hit would not be very high.

Of course, these solutions come with a cost. It makes sense to find out if it is needed to use them, or if the resulting synchronization is good enough for a given scenario. To get an idea of what is possible, we also implemented the protocol with an explicit indication sender, that is, a station which only sends out indication messages. This way, the master physically receives the indication, as all the slaves, and behaves the same way as these. More or less, this mimics the scenario when the master and the slaves do not use buffering for sending and receiving. The measurements will show if there is a significant improvement in the precision of the protocol.

Another option for improvement is to make the wireless network card itself take the timestamp on the master and on the slave. This removes all operating system time and such details like interrupt processing from the critical path. Only timing issues inside the network card would be relevant. Of course, the network card itself can be controlled much better than the whole host machine. Thus, most probably, this would increase the precision very much.

For more details on the protocol, and how it handles omissions, we refer the reader to [5].

3.2. Continuous clock correction

Instead of applying the clock difference directly to the slave's clock, the clocks are adjusted with the help of two clock differences at two instances of time. This way, the precision can be improved as well as a continuous clock can be achieved. Furthermore, the protocol profits if information about clock differences is known which are farther apart, allowing to lower the frequency the time information is sent. This reduces the com-

putational and network overhead significantly, allowing for power-constrained operation needed in sensor-networks.

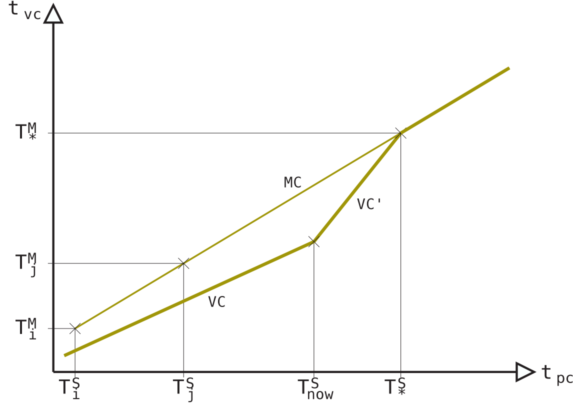

Figure 3. Calculation of the global clock

Figure 3 shows the clock correction as done by the protocol on one slave. On the x-axis, the physical clock of the slave is shown, while on the y-axis, the virtual clock of the slave is diagrammed.

Assume that two synchronization rounds i, j, with $i \neq j$, took place in the past. Now, according to the above protocol, the slave knows the following facts: At its physical time T_i^S, the master's clock was T_i^M. Additionally, at its phyiscal time T_j^S, the master's clock was T_j^M.

According to this, the client can calculate the master's clock as seen by him. It calculates the straight line through (T_i^S, T_i^M) and (T_j^S, T_j^M) (MC in figure 3). Assuming it is now T_{now}^S at its physical clock, and the current clock of the slave is VC, it could adjust its virtual clock to have exactly the gradient and the offset of the sender's clock, resulting in changing from VC to the clock MC at time T_{now}^S.

Of course, adjusting the time directly leads to time jumps. To avoid this, instead of using the master's clock directly as its own virtual clock, the slave adjusts its clock in two steps. In the first step, it adjusts its clock in such a way that it smoothly approaches the master's clock at some instance of time T_*^S in the future. This results in the clock VC', as seen in figure 3. After T_*^S, it uses the master's clock as its own virtual clock. This way, a continuous clock can be achieved very easily.

This protocol needs a master to synchronize slaves in direct reach. It can be easily extended to synchronize a whole sensor network as follows: The sensors build clusters, with a master as clusterhead being able to reach all of its cluster members directly, running the protocol. Slave stations which can be reached by more than one master synchronize to both masters, remembering the time difference between both clusters. Whenever these slaves route time information from one cluster to the other, they adjust the times with their knowledge of the time in the different clusters.

For more details on the protocol, we refer the reader to [5] and [6].

4. Implementation

The clock synchronization protocol has been implemented on RT-Linux/Free, a GPL real time extension to the Linux kernel based on the work of FSMLabs Inc. RT-Linux was chosen because of its real time capable interrupt request (IRQ) management, which is best suited for optimal synchronization precision.

The protocol has been integrated into RT-Orinoco, our RT-Linux implementation of the Orinoco driver, but the driver modifications were limited to several key positions, so a migration to other hardware drivers or to a more generic API (application programming interface) would be easily possible.

Depending on the role of the station (master, client or indication server) the driver has to complete different tasks. For most of them, a precise timestamping of the Orinoco hardware interrupt is needed. To accomplish this, the IRQ handler has been extended by a query of the current local time as its first action. The so gained time stamp is then used when a master or indication packet is received, or when the master finishes sending.

A procedure has been added to the beginning of the RT-Orinoco IRQ handler to get the local timestamps of global events (i.e. the reception of indication packets). An additional out-of-band packet buffer was added into the driver for the transmission of indication packets, and a periodic master thread has been implemented to generate such packets.

The virtual clock is based on the RT-Linux `gethrtime()` function on the master machine. In the x86 implementation of RT-Linux, this function is based on the `rdtsc` command of the CPU, which has a granularity of several nanoseconds. Thus, it is suitable for the synchronization protocol.

The driver also has been extended by an additional packet buffer for out-of-band packets. This buffer is used for time synchronization packets, because of their higher priority status and because they must not be retransmitted in the case of an error.

An additional thread is running on the indication server (or on the master, when separate indication is

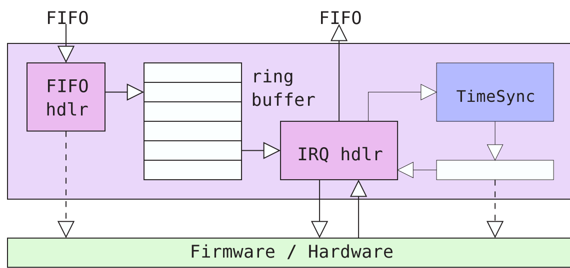

Figure 4. Structure of the modified RT Orinoco driver

not activated), which is periodically scheduled to send out broadcast packets. Raw Ethernet frames with a distinguished packet type are used for the synchronization. These frames are automatically converted to the right 802.11 frame format by the Orinoco firmware. To accomplish application transparency, the synchronization packets are generated in the driver when sending, and the driver analyzes them and inhibits their transfer to the higher level application on reception.

The transformation of local to virtual time and vice versa requires several additions and a multiplication, which has been implemented as 64bit x 32bit with a 64bit result. To calculate a new virtual clock with rate adaptation, several additions, shifts and a 64bit division are needed. These operations could be reduced to 32 bits for better calculation performance on sensor devices when using a master clock with a bigger granularity. Note that all calculations are done with fixed point arithmetic.

5. Evaluation

5.1. Structure

To measure the clock differences between different WLAN stations it was needed to connect them all to one single observer machine. The communication has been implemented as signal edges on the parallel ports of the participating machines, which were all connected to the input pins of the observer parallel port. The observer, a Pentium 133 machine in real mode, is busy waiting for changes on the port and logging them using its `rdtsc` time stamp. This construction allows a measuring precision of $5\mu s$ and the results can be compared to the ones achieved in [6], as the same setup was used. The synchronization protocol is deployed on four Athlon XP 2400+ machines with Orinoco PCMCIA wireless network devices.

To measure the precision of the virtual clocks, an additional RT-Linux task is run, which toggles one of the parallel port bits in periodic time intervals based on the virtual clock. This allows the observer to compare the virtual clocks of all participating machines and to see their differences.

To reach maximal precision in the task an approach of periodic clock polling has been chosen, which switches to busy waiting when the trigger interval is almost over. Such an energy consuming procedure is not needed when only recording the timestamps of events, it is only required for events triggered by the virtual clock. A scheduler extension is planned to fix this problem by utilizing the global clock for waking up threads.

To see the influences of a dedicated indication station and of traffic load on the medium, four different tests have been performed. Additionally, the reliability of the protocol has been tested in the case of a master failure.

As the total number of participating machines was four, with explicit indication there were two clients (plus master and indication server), while without explicit indication there were three client machines.

The diagrams in figures 5-10 display the trigger time difference of the clients compared to the master's virtual clock, as measured by the observer. We used a round time of 3s for these measurements. Please take note that the diagrams are heavily disproportional due to the fact that the time difference on the Y axis is in the magnitude of microseconds whereas the X axis is several minutes long.

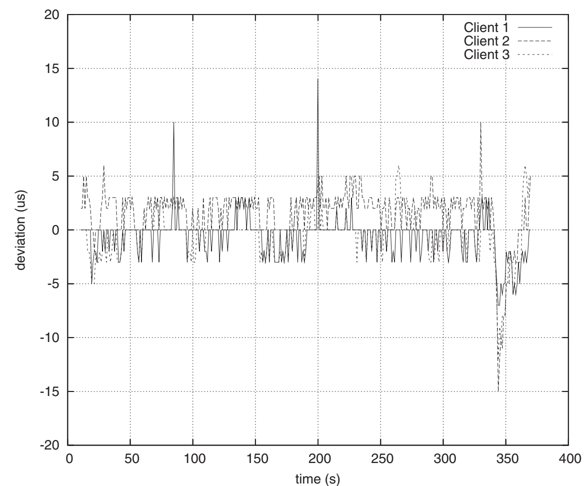

Figure 5. Measured synchronization precision: no traffic

The network load was simulated by another two machines in the same Ad-Hoc wireless cell, which were generating continuous bidirectional TCP traffic using

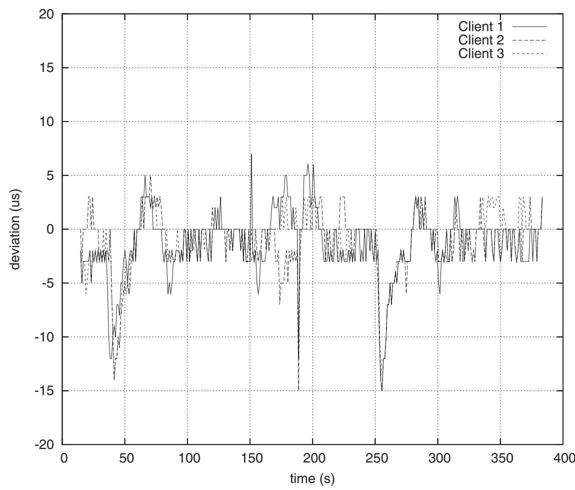

Figure 6. Measured synchronization precision: 650KB/s

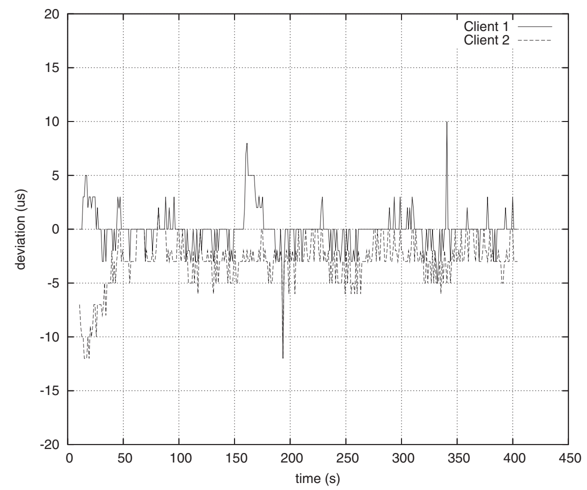

Figure 7. Indicated synchronization: no traffic

the Linux tool `iperf -d`, leading to a load of approx. 650 KByte/s.

5.2. Results

Figure 5 shows the results which are observed if no additional traffic is put on the network. A short time after the clients synchronize with the master's virtual clock, their accuracy stays below $15\mu s$, that is, at all times, they are not more than $15\mu s$ away from the master clock. Note that most of the values are even in the magnitude of the observer's resolution.

When the medium is under traffic load (diagrammed in figure 6), it can be seen that the clients tend to be a bit faster than the master in several places. This is due to a delay in the master's transmit IRQ, which is caused by the Orinoco card firmware. Using a station as a timestamp indicator is a workaround for this problem.

When deploying an explicit indication message, the master and the clients are using the reception of the same packet as a common trigger event, so the traffic load has no negative impact on the virtual clock precision. Both with (figure 8) and without (figure 7) traffic load, the precision is better and the clocks are steadier than without indication, though again the difference of the clients to the master is not much greater than the measuring resolution.

When the master machine, being the central synchronization instance, drops out, the clients use the information from the last rounds to adapt their virtual clocks. This can be seen in figure 9 for the case without traffic, and in figure 10 for the case with traffic. Because this information is not perfect and there are no updates, the clients drift away from the master and leave the regular precision window after several minutes. Anyway, in both cases, even after $300s$, all clients are not more than $20\mu s$ away from the server, which shows that the clock rate adjustment worked very well.

Additional tests have shown in practice, a master failure (like a continuing disturbance of the WLAN medium) can be tolerated for over $200s$ without the clients leaving the $15\mu s$ accuracy corridor. Thus it is possible to increase the round time from $3s$ used in the tests to $30s$ or even $60s$ to further decrease the energy consumption of the protocol without losing precision. With longer round intervals the rate adoption algorithm is even expected to become more precise due to a better time interval to jitter ratio. We are working on evaluating this.

6. Conclusion

We presented an evaluation of a clock synchronization protocol which is suited for sensor networks. It profits from low frequencies of synchronization rounds, which reduces network utilization significantly. Furthermore, this protocol allows for adjusting clock values across broadcast domains, making it well-suited for sensor-networks. We will evaluate this more thoroughly in some following work.

References

[1] Ö. Babaoglu and R. Drummond. (almost) no cost clock synchronization. In *Proc. 17th Annual International*

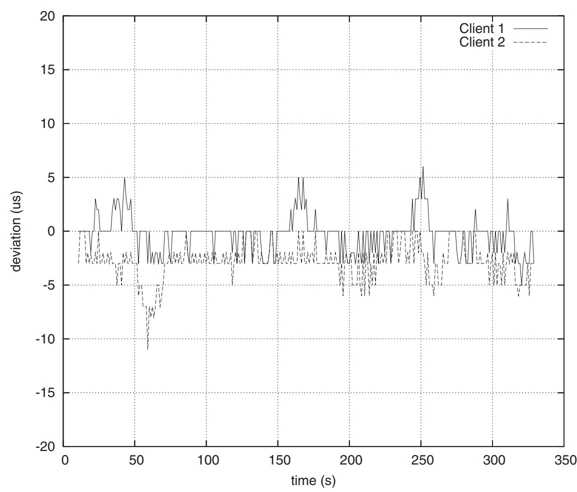

Figure 8. Indicated synchronization: 650KB/s

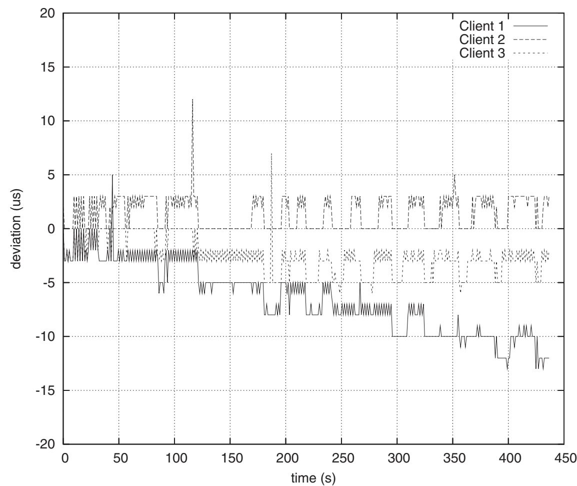

Figure 9. Master failure: no traffic

Symposion on Fault-Tolerant Computing (FTCS 87), pages 42–47, July 1987.

[2] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *Proc. Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, volume 36, pages 147–163, 2002.

[3] M. Gergeleit and H. Streich. Implementing a distributed high-resolution real-time clock using the canbus. In *1st international CAN-Conference*, 1994.

[4] J. Lundelius and N. Lynch. An upper and lower bound for clock synchronization. In *Information and control*, volume 62, pages 190–204, 1984.

[5] M. Mock, R. Frings, E. Nett, and S. Trikaliotis. Clock synchronization for wireless local area networks. In *12th Euromicro Conference On Real Time Systems (ECRTS)*, pages 183–189, June 2000.

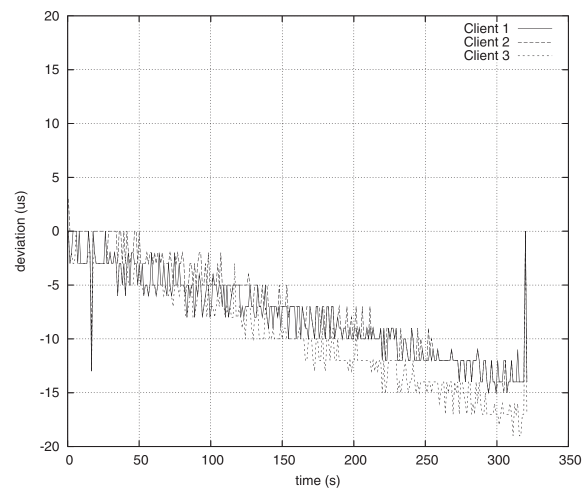

Figure 10. Master failure: 650KB/s

[6] M. Mock, R. Frings, E. Nett, and S. Trikaliotis. Continuous clock synchronization in wireless real-time applications. In *Symposium on Reliability in Distributed Software (SRDS)*, pages 125–132, Oct. 2000.

[7] B. Sundararaman, U. Buy, and A. D. Kshemkalyani. Clock synchronization in wireless sensor networks: A survey. In *Ad-Hoc Networks*, volume 3(3), pages 281–323, May 2005.

[8] P. Veríssimo and L. Rodrigues. A posteriori agreement for fault-tolerant clock synchronization on broadcast networks. In D. Pradhan, editor, *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing (FTCS 92)*, pages 527–536. IEEE Computer Society Press., July 1992.

[9] Y. Zhao, W. Zhou, J. Huang, S. Yu, and E. J. Lanham. Self-adaptive clock synchronization based on clock precision difference. In *Proceedings of the Twenty-Sixth Australasian Computer Science Conference (ACSC2003)*, pages 181–187, Feb. 2003.