

# Realization of a Distributed OS Component for Internal Clock Synchronization in a LAN Environment

K. H. (Kane) Kim, Chansik Im, and Prasad Athreya

University of California  
Irvine, CA, USA

{khkim, imc, nathreya} @uci.edu, <http://dream.eng.uci.edu/>

**Abstract:** A key challenge in keeping distributed clocks well synchronized in a LAN environment via OS- or middleware-level mechanisms is to create an interference-free condition in a network channel(s) and a non-preemptible on-alert condition in the responsible thread in each node during each resynchronization round while keeping the length of each round as short as possible. An approach for meeting this challenge which is based on the middleware architecture TMOSM, has been presented. A prototype implementation and its measurements indicated that with a typical network of PCs connected via a 10 Mbps Ethernet, the PC nodes can be kept synchronized within the deviation bound of 0.5 milliseconds. The organizing principle is believed to be applicable to many real-time OSs or middleware used to support distributed computing applications.

**Keywords:** real time, clock, synchronization, deviation, embedded, TMO, time triggered, message, object, middleware, distributed, parallel, programming.

## 1. Introduction

A high-precision time base accessible from everywhere in a networked computing environment is essential in realizing reliable real-time (RT) distributed computing. Such a time base is established by synchronizing the local clocks of distributed computing nodes among themselves and also synchronizing them to a national or international standard time source such as UTC (Universal Time Coordinated) [Kop97]. When all the member nodes of a distributed computing system (DCS) are equipped with GPS (Global Positioning System) receivers, their clocks can be synchronized to UTC with the maximum deviation of a few microseconds [Tur02]. In the environments, where only a small subset of the computing nodes are equipped with GPS receivers, other nodes must synchronize their clocks to those of the nodes equipped with GPS receivers. This case of synchronization is called the *internal synchronization* [Kop97]. It is well known that local clocks in distributed computing nodes will diverge since they rely on physical oscillators which tend to drift away from each other. Our experiments show that local clocks of two PCs could become as much as 100 microseconds apart from each other in one second. Therefore, distributed clocks should be resynchronized periodically in order that the differences among the distributed clocks may stay

bounded within a specific threshold.

Resynchronization of distributed clocks is an extensively studied subject [Kop97, Ger94]. Nevertheless, its efficient implementation in a local area network (LAN) environment where member nodes are common PC stations without special hardware components for clock synchronization or similar computing devices, is not a well established engineering branch. For example, the current most popular OS used in desktop or notebook-size computing stations, the Microsoft Windows OS family, uses the NTP (Network Time Protocol) [Mil91, NTP02]. NTP is used to synchronize the time of a computer client or server over the Internet to that of another server. However, NTP alone is not really meant for use in high-reliability RT DCSs because NTP itself is not concerned with the need for yielding any determinable bounds on the precision of the global time base realized, i.e., the bounds on the deviations among the distributed clocks.

In this paper, we discuss a fundamental issue that is encountered in periodically (or even sporadically) resynchronizing distributed clocks by use of OS-level or middleware-level protocols in an RT DCS. The issue is how to minimize or eliminate unpredictable interferences on the network through which resynchronization messages are transmitted during the resynchronization period. If the message communication delay through the network is highly deterministic, then clock resynchronization becomes a relatively easy problem.

In the course of our recent development of a middleware system supporting the execution of distributed RT objects, we identified an approach to structuring and operating the middleware which leads to elimination of unpredictable network interferences during clock resynchronization, thereby leading to a relatively high precision of the global time base realized. The principle of organizing such middleware and the measured performance data from a prototype implementation of the clock resynchronization scheme are presented in this paper. The organizing principle is believed to be applicable to many RT OSs or middleware used to support distributed computing applications.

In the next section (2), an overview of the middleware architecture devised to support RT distributed computing objects is given. Section 3 then discusses the middleware-level approach for facilitating interference-

free sharing of LAN facilities. The issues in clock resynchronization and the developed resynchronization scheme are presented in Section 4 and Section 5.

## 2. An overview of the TMO Support Middleware (TMOSM)

The middleware architecture we have developed to support execution of RT distributed computing objects has been named the *TMO support middleware* (TMOSM). The RT distributed objects supported are *Time-triggered Message-triggered Objects* (TMOs) [Kim97, Kim00]. TMO is a natural, syntactically minor, and semantically powerful extension of the conventional object(s). Significant extensions are summarized below.

- (a) Distributed computing component; abstract styles of remote method calls and easy migration among the computing nodes.
- (b) Clear separation between two types of methods, *time-triggered methods* (also called *spontaneous methods* or SpMs) and *service methods* (SvMs); The SpM executions are triggered upon reaching of the real-time clock at specific values determined at the design time whereas the SvM executions are triggered by service request messages from client objects.
- (c) *Basic concurrency constraint* (BCC) which protects SpM executions in using needed resources against interferences from SvM executions;
- (d) *Guaranteed completion time* of the server (i.e., an SvM of a server TMO) and the *result return deadline* imposed by the client.

To support the execution of TMO-structured programs, the underlying execution engine should possess the following capabilities:

- (a) High-precision timer service;
- (b) Concurrent, preemptive execution of SpMs and SvMs;
- (c) Timely initiation and completion of SpM and SvM executions.

First of all, RT applications, including TMO-structured applications, need to initiate or complete some actions at specified real time instants. Therefore, a high-precision timer, which can be queried by the application, should be provided by the execution engine. Concurrency within each TMO and among TMOs must be exploited to the maximum practical extent. The underlying execution engine must thus schedule multiple threads/processes (assigned to execute TMO parts) concurrently as well as preemptively with given urgency values reflected.

The TMO execution engine can be implemented by extending an OS kernel or adding middleware that wraps around an OS kernel. It should provide an easy-to-use programming interface to the TMO programmers. Building a new RT OS kernel to support TMO programming can lead to realization of better

performance. On the other hand, by using the middleware implementation approach, TMO programming with a large variety of existing COTS platforms such as Windows or Linux can be facilitated. TMOSM is a TMO execution support middleware model which can be easily adapted to most COTS platforms [Kim99].

The internal thread structure of TMOSM is shown in Figure 1. This architecture has been devised to enable relatively easy analysis of the worst-case execution times of TMO methods. TMOSM consists of three types of threads, *application threads*, *middleware threads*, and the *super-micro thread*. TMOSM assigns one application thread to each execution of a method (SpM or SvM) of an application TMO. Middleware threads are *periodic threads* (periodically activated to run for a time-slice), each being responsible for a major part of the functions of TMOSM. The authors believe that structuring of middleware threads as periodic threads is a fundamentally sound approach which leads to easier analysis of the worst-case time behavior of the object execution engine without incurring any significant performance drawback [Kim99]. The super-micro thread is called the WTST (*Watchdog Timer & Scheduler Thread*). It is a "super-thread" in that it runs at the highest possible priority level on the node. It is also a "micro-thread" in that it manages the scheduling / activation of all other threads in TMOSM. WTST is activated whenever a thread switching needs to be performed, e.g., upon expiration of a time-slice. Even those threads created by the node OS before TMOSM starts are executed only if WTST allocates some times-slices to them. Therefore, WTST is the best candidate to be responsible for clock resynchronization since no other threads in a node can preempt WTST except OS interrupt handlers [Sol00]. Also, WTST checks for any deadline violations and if a violation is found, it provides an exception signal to a relevant computation unit.

The three middleware threads function as follows:

(1) MMCT (Middleware Message Communication Thread): This periodic thread manages the sending of *middleware messages* through the communication network. Middleware messages are the messages exchanged among the middleware instantiations running on different nodes to support interaction among TMOs. MMCT also distributes middleware messages coming through the network to their destination threads.

(2) VMST (Virtual Main System Thread): Periodically a time-slice is conceptually given to this virtual thread which merely represents all application threads running TMO methods. The actual time-slice allocations are taken by WTST that executes the application scheduler function. Every time-slice conceptually belonging to the VMST is allocated to a fairly selected application thread.

(3) VIST (Virtual I/O System Thread): This virtual thread maintains a pool of threads which are called *Local I/O system threads* (LIITs) and execute the I/O requests

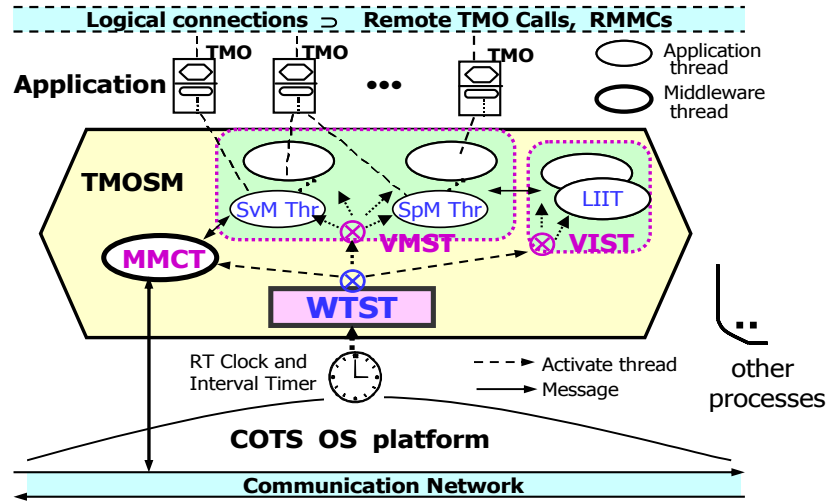


Figure 1. The basic internal thread structure of TMOSM (Adapted from [Kim99])

from application threads. The time-slices allocated to VIST are actually distributed to LIITs. Each LIIT is assigned to execute an I/O function that utilizes the I/O capabilities of the host node platform including serial character I/O, disk I/O, network I/O involving messages which are not middleware messages, etc. This VIST approach has been motivated by the desire to make it easier to analyze with high precision the temporal predictability of application program-segments not involving I/O (and, to a less extent, the temporal predictability of I/O activities).

Several features of this TMOSM architecture contribute to simplifying the analysis of the execution time behavior of application TMOs running on TMOSM. First, the strictly periodic nature of middleware threads and the dedication of each middleware thread to a specific functionality enable largely independent analysis of the part of the execution time behavior of application TMOs that depends on a particular middleware thread. For example, in computing the maximum time taken for transmitting a message  $\alpha$  in a queue attached to MMCT to a queue attached to the MMCT in a remote node, one needs to focus on a few factors only: the number and sizes of the messages in the queue ahead of  $\alpha$ , the size of  $\alpha$ , guaranteed bandwidth of the network path between the source and the destination, and the frequency and the size of the time-slice given to MMCT in each node. Secondly, the execution time of an I/O can be in general specified, analyzed, and measured in a larger time gain than that which can be used in specifying and analyzing the computation relying on the CPU only. Therefore, dedicating LIITs to handling such I/O activities enables the high-precision analysis of the execution time behavior of the CPU-intensive computation.

TMOSM provides an interface, `GetCurrentDCSage()`, which can be called by the application TMO programmer

to get time information. This API will return the current *distributed computing system age* (DCSage), which starts from zero at the initialization of TMOSM. In the current implementation of TMOSM on Windows XP/2000/NT, the resolution of DCSage TMOSM pretends to support is 1 microsecond but to the application TMO programmer, the precision of the time value that can be trusted is at the level of a millisecond.

### 3. Cooperative TDMA access to the LAN facility by TMOSM instantiations

The communication delay among different nodes should be bounded for proper operation of TMO-based applications. It is necessary to guarantee the completion of client-server transactions between TMOs within their corresponding deadlines and to guarantee the arrivals of other middleware messages at their destinations on time. However, in an open unregulated network environment, such as Internet, the communication delay is not bounded. Moreover, to guarantee the arrivals of clock resynchronization messages at their destinations within predictable bounds, bounded and predictable communication delay with an acceptable level of communication jitter is of critical importance.

To make communication delay to be bounded and communication jitter to be minimized, a preferred communication network environment where TMOSM runs is the following LAN environment that is isolated and regulated. First of all, the LAN to which computing nodes hosting TMOSM instantiations are connected is isolated from the outer world such as Internet and only those distributed computing nodes which collectively run a TMO-based RT distributed application use the LAN. For regulating the access to the communication network channel, participating TMOSM instantiations access the

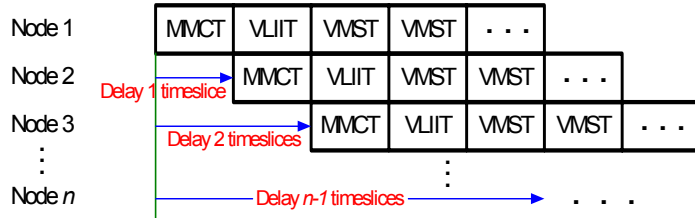


Figure 2. Staggering of MMCTs in different nodes

communication channel in a TDMA (Time Division Multiple Access) fashion. The TMOSM instantiation in each participating distributed node sends out messages to the communication network only when the middleware thread MMCT within the TMOSM instantiation is executed. Moreover, middleware threads in every node have been arranged in the same order for receiving time-slices. By synchronizing time-slices among the participating nodes and staggering the time-slices allocated to MMCTs in participating nodes, we can ensure that at any give time, only one node will run its MMCT. Figure 2 shows staggered MMCTs among  $n$  participating nodes.

Also, before every MMCT accesses the communication network channel, it must wait for a short interval to make sure that the messages sent by MMCT of another node during the preceding time-slice have been delivered to the corresponding destination completely and there is no message on the channel. Again, at any given time, only one node accesses the communication network channel. This means that the possibility of message conflicts is minimized and a tight communication delay bound can be determined.

Figure 3 compares round-trip communication delays in a non-regulated network environment against those in a regulated network environment. According to the figure, 83% of the round-trip communication delays are bounded within the range from 210 microseconds to 230 microseconds when the communication network is

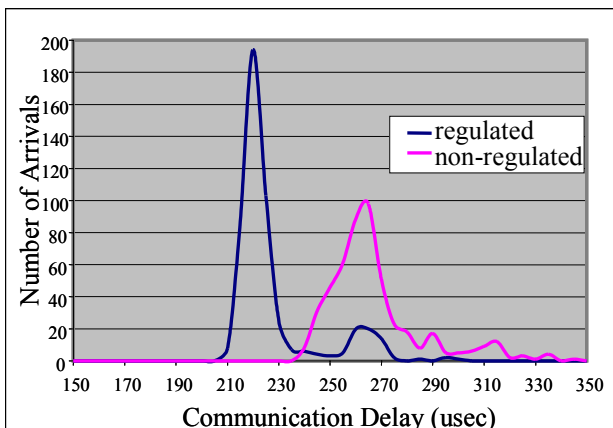


Figure 3. Round-trip communication delays

isolated and regulated. The additional delays beyond 230 microseconds are mainly due to the unpredictable part of the OS and protocol software overhead.

#### 4. The scheme adopted in TMOSM for initial clock synchronization

During the initialization phase of a TMO-structured DCS, one or more nodes instantiate TMOSM(s). One of those nodes is the *master node* selected and configured as such at the design time. To be more specific, each node contains a file named “config.ini” where the master or worker status of the node is indicated. As a worker TMOSM is instantiated, it registers itself with the master TMOSM instantiation. When the master TMOSM realizes that an expected number of worker TMOSMs have registered, it initiates the *initial clock synchronization*. When the clock synchronization is over, every TMOSM instantiation assumes that cooperative distributed computing just started and thus it records its local clock value as that corresponding to the beginning of DCSage. In this section, the scheme adopted for this initial clock synchronization is discussed. Another kind of clock synchronization performed by TMOSM instantiations is the *periodic clock resynchronization*. It will be discussed in Section 5.

##### 4.1. Master-worker scheme

One of the simplest ways to achieve internal clock synchronization is to adjust a local clock of every participating node to the local clock of a specific node selected to be the master node. Figure 4 depicts a simple master-initiated internal clock synchronization scheme. At time  $t_1$ , the master takes a current timestamp based on its local clock. Then the master node broadcasts the timestamp as a clock synchronization message at time  $t_2$ . The clock synchronization message will arrive at each worker node ( $t_3, t_3', \dots$ ) and each worker node takes the timestamp based on its own local clock upon receiving the clock synchronization message (an event occurring at time  $t_4$ , or  $t_4'$ , etc.). Then each worker node will determine the difference between the master clock and its own local clock based on the timestamps. Finally, on the basis of this information, worker nodes adjust their local

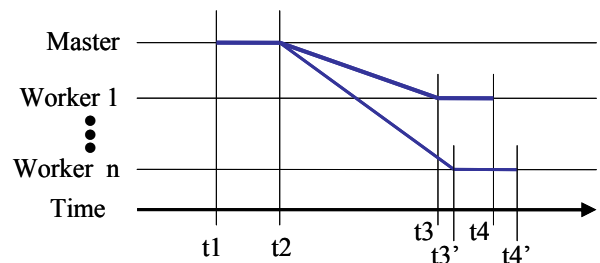


Figure 4. A simple master-initiated clock synchronization scheme

clocks.

This protocol is quite simple and easy to implement. However, it must be executed while maintaining the system in an interference-free environment as much as possible. For example, processes/threads, which are responsible for taking, sending, and receiving timestamps, should not be preempted during the clock synchronization. Also, communication jitter should be suppressed as much as possible during the clock synchronization. The IEEE 802.11 standard for wireless communication in LAN provides a clock synchronization protocol similar to this simple protocol [IEE97].

#### 4.2. Initial clock synchronization in TMOSM

Normally, all TMOSM instantiations are initialized together before a TMO application starts, actually before the master TMOSM initiates the clock synchronization. However, sometimes certain TMOSMs can be initialized late after the DCS initialization (*late initialization*). For example, if one node becomes faulty and detected, it will be shut down. Some time later, the faults have been eliminated and the node is resurrected to rejoin into a TMO execution network. TMOSM in the rejoining node should be initialized during resurrection. Also, DCSage of such a node should be set correctly during the TMOSM initialization. The clock of a node being resurrected is synchronized to that of the master during the earliest arriving resynchronization period. Initial clock synchronization is done by a special thread named AIT (*application initialization thread*). AIT is the thread that executes the main function Main() in a C++ TMO program. The first statement in Main() is usually StartTMOengine() and this latter function creates a TMOSM instantiation, registers the created TMOSM instantiation with the master TMOSM if the created TMOSM is a worker TMOSM, and performs the initial clock synchronization. So during the initial clock synchronization, AIT is the only active thread within the node and has the highest priority.

The communication delays from the master node to worker nodes can be estimated reasonably by using measurements of round-trip communication delays. Actually, such communication delays in an isolated and regulated small-area LAN environment are likely to be constants. Moreover, if the LAN distance is limited to 100 meters or less, it is not worth differentiating such communication delay estimates for different workers. Therefore, communication delay is measured once or twice only between the master node and one selected worker node during the TMOSM initialization. Then, this value will be used as an expected communication delay for a clock synchronization message. In this way, only one broadcast message is required per each round of clock synchronization.

During the DCS initialization stage, every worker node sends a registration message to the master node and waits for a TMOSM\_STARTUP message from the master node. When the master node receives all registration

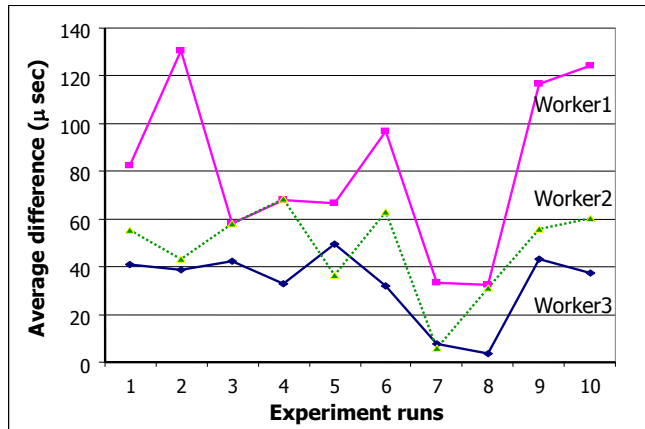


Figure 5. Initial clock synchronization

messages from all expected distributed TMOSMs, it picks up one specific worker node as a “representative” worker node. A round-trip communication delay ( $\Delta$ ) between the master node and the representative worker node is measured. After obtaining the communication delay measurement, the master node broadcasts a TMOSM\_STARTUP message, which contains an *one-way communication delay estimate*,  $\Delta/2$ , to all waiting worker nodes. It then sleeps for  $\Delta/2$  and sets the “local view of the DCSage” to zero. This completes the execution of StartTMOengine(). Each worker node, as soon as a TMOSM\_STARTUP message arrives, sets the local view of the DCSage to zero and save  $\Delta/2$  for future use.

If a TMOSM instantiation is created late or resurrected after experiencing a failure, it sends a registration message to the master node and waits. The master node responds with a TMOSM\_WELCOME message containing its view of the current DCSage and the one-way communication delay estimate  $\Delta/2$ . When the worker node receives the TMOSM\_WELCOME message, it sets its view of the DCSage to the master’s view plus  $\Delta/2$  and waits until the next clock resynchronization period arrives. Then the worker node performs a clock resynchronization.

Figure 5 shows the average of the differences between the view of DCSage held by the master node and that held by each of the three different worker nodes after the initial clock synchronization has been performed. For this experiment, four Pentium III 500Mhz or higher-speed PCs were connected to a 10Mbps Ethernet LAN. Just after the initial clock synchronization had finished, the master node sent the current DCSage to the worker nodes every 1 millisecond. When worker nodes received the message from the master node, they took their local views of the current DCSage and calculated the differences in views of the DCSage. The one-way communication delay, which had been estimated during initial clock synchronization, was reflected in calculating the differences. Total 10 messages were sent out during each

experimental run and the average deviation of each worker's view from the master's view of the DCSage was calculated. The results of 10 experimental runs were collected and they are shown in the figure.

## 5. Issues in and approaches for clock resynchronization in TMOSM

### 5.1. Master-worker resynchronization

Due to the differences in the clock drift rates of participating distributed computing nodes, the local view of the DCSage held in each node will diverge from the views of others after initial clock synchronization. Therefore, the clocks of distributed computing nodes must be resynchronized periodically. Again, the master-worker scheme, which was already described in Section 4.1 was adopted. To prevent the thread responsible for clock resynchronization from being preempted, WTST, which has the highest priority in each node, is used to perform clock resynchronization. WTST in each node enters the *clock resynchronization mode* based on the local view of the DCSage.

### 5.2. Silent period

Unlike the initial clock synchronization, the clock resynchronization problem poses an additional challenge of arranging for nearly simultaneous entries of the distributed computing nodes into the resynchronization mode. This is important because if the master suddenly broadcasts its current time, the delay incurred before each worker node, which had been busy executing application TMOs, recognizes the message, will vary widely among the worker nodes. Therefore, the master node must try to broadcast its view of the DCSage when all worker nodes are ready to examine the messages coming from the master node with minimal delays and the network channels over which clock resynchronization messages will be transmitted are interference-free, i.e., silent. All participating nodes including both the master node and worker nodes will enter the clock resynchronization mode at  $T_{nr} - \Pi - \epsilon$ , where  $T_{nr}$  is the time for the next clock resynchronization,  $\Pi$  is the maximum deviation possible between the local clock of the master node and those of worker nodes, and  $\epsilon$  is the maximum length of the period in which a local network interface card (NIC) can autonomously access the communication channel (e.g., Ethernet) before becoming idle. During clock resynchronization, all threads but WTST will be suspended. Then all tasks normally carried out by WTST but the clock resynchronization task will be suspended. The master node will wait until  $T_{nr}$  is reached and then it will broadcast its view of the current DCSage as a clock

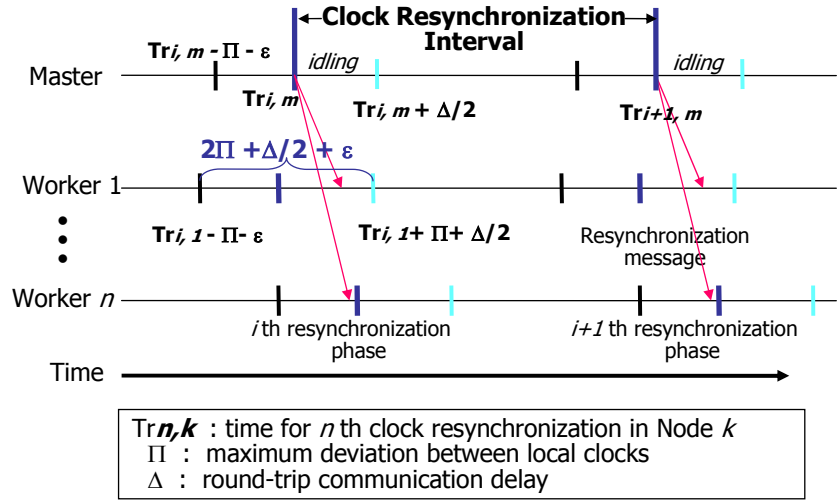


Figure 6. Clock resynchronization rounds

resynchronization message, sit idling for  $\Delta/2$ , and exit from the clock resynchronization mode. Worker nodes will wait for the arrival of the clock resynchronization message or the *Resynchronization Timeout* ( $2\Pi + \Delta/2 + \epsilon$ ) event.

Figure 6 shows how the silent periods are realized during the clock resynchronization mode. The figure shows that Worker1 may wait for a clock resynchronization message from the master until  $Tr_{i,1} - \Pi + \Delta/2$ , which is  $\Pi + \Delta/2$  later than  $Tr_{i,1}$ , i.e., the time at which Worker1 expects the master to send the  $i$ -th round clock resynchronization message out. The factor  $\Delta/2$  here is incorporated because that is the expected travel time of the message. The factor  $\Pi$  is there because  $Tr_{i,1}$  can be  $\Pi$  earlier than  $Tr_{i,m}$ , i.e., the time at which the master node sends the  $i$ -th round clock resynchronization message out. Therefore, a worker node may be in the clock resynchronization mode for as long as  $2\Pi + \Delta/2 + \epsilon$  units of time.

If a worker node receives the clock resynchronization message, it time-stamps its view of the current DCSage. Then it calculates the difference between the master's view and its own view, *clock\_diff*. The one-way communication delay estimate obtained during the initial clock synchronization will be reflected in calculating *clock\_diff*. If its clock is slower than the master clock, the worker node simply adjusts its view of the DCSage and lets it jump forward by *clock\_diff*. If its clock is faster than the master clock, its view of DCSage is set to jump back by *clock\_diff* and its TMOSM instantiation will be halted for a time equal to *clock\_diff* to prevent TMO-based applications from experiencing the arrival of the same real-time instant twice.

One thing important here is that application TMO designers must understand the fact that each node running TMOs will periodically enter the clock resynchronization mode, which is practically a hibernation mode as far as

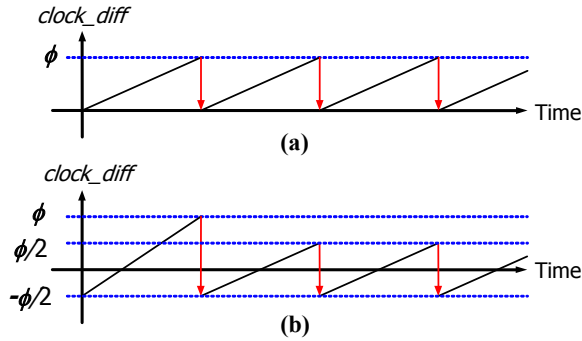


Figure 7. Deviation of the worker's clock from the master's clock

application software and the rest of the OS are concerned. Therefore, if the resynchronization occurs once per second and  $2\pi + \Delta/2 + \epsilon$  is 1.5 milliseconds, the computing node practically disappears for up to 1.5 milliseconds once per second. The application designer may want to reflect this knowledge in designing each TMO and in some cases, the designer may want to know the time-windows for resynchronizations in calendar time to the precision of 1/10 to 1/2 of a time-slice.

### 5.3. Optimal clock adjustment when drift rates are steady

In typical computing nodes, clock drift rates tend to be steady. This means that if the view of the DCSage in a worker node is adjusted to that of the master node in each resynchronization round, the deviation of the worker's view from the master's view will show the pattern exhibited in Figure 7(a). In the figure,  $\phi$  represents the maximum measured deviation just before a clock resynchronization.

Since the purpose of clock resynchronization is to make the differences among the DCSage views of participating distributed nodes to be bounded within the smallest possible range, it is worthwhile trying to make the maximum deviation of each worker's view from the master's view to be as small as possible. As long as clock drift rates are steady, the approach depicted in Figure 7(b) is advantageous. As shown, the view of the worker (i.e., the local clock) is adjusted in the first resynchronization round to jump backward (in this case while jumping forward in the opposite case) by  $(3/2)*\phi$  rather than  $\phi$ . Then, at the next clock resynchronization point, the measured deviation will be  $\phi/2$ . This time and in each of the future resynchronization rounds, the local view of the worker is adjusted to jump backward by  $\phi$ . The deviation will then always range between  $-\phi/2$  and  $\phi/2$ , except during the period between the initial clock synchronization and the first resynchronization. This is a significant improvement over the situation depicted in Figure 7(a).

Further improvement can be achieved by performing the first resynchronization round only one half of the normal inter-resynchronization interval after the initial

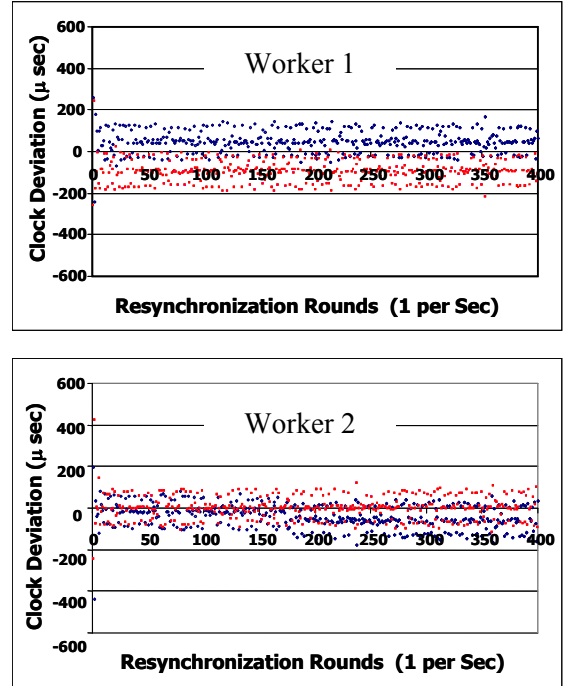


Figure 8. Clock deviations immediately before and after clock resynchronizations

synchronization and doing every subsequent resynchronization round a full normal inter-resynchronization interval after the preceding round. This will make the deviation to be always within the range of  $-\phi/2$  to  $\phi/2$ .

Again, four Pentium III 500Mhz or higher-speed PCs connected to a 10Mbps Ethernet LAN were used for evaluating the clock resynchronization schemes discussed here and in the preceding subsection (5.2). To measure the clock deviation between the master node and the worker node after a clock resynchronization, each worker node sends a message back to the master node as soon as a clock resynchronization is done and waits for a reply from the master. The master node waits until messages from all worker nodes arrive. When the master node receives all messages, then it broadcasts its view of the current DCSage to all worker nodes. Worker nodes timestamp their views of the DCSage when they get a reply from the master node and compute clock deviations while reflecting the one-way communication delay estimate obtained during the initial clock synchronization.

Figure 8 shows the measured quality of clock resynchronization realized. Each black dot represents the difference in the view of DCSage between the master node and the worker node just before a clock resynchronization. Each gray dot represents the difference just after a clock resynchronization. The clock adjustment approach in Figure 7(b) was used but the additional improvement possibility mentioned above for keeping the deviation always between  $-\phi/2$  to  $\phi/2$  was not

implemented at the experimentation time. As shown in the figure, the clock deviations are bounded within 500 microseconds at the first clock resynchronization. After the first round, the clock deviations are bounded within 200 microseconds.

## 6. Conclusion

A practical scheme for keeping distributed clocks well synchronized in a LAN environment via OS-level mechanisms has been presented in this paper. In our view, the key challenge is to create an interference-free condition in a network channel(s) and a non-preemptible on-alert condition in the responsible thread in each node during each resynchronization round while keeping the length of each round as short as possible. An approach for meeting this challenge which is based on the middleware architecture TMOSM, has been presented. A prototype implementation and its measurements indicated that with a typical network of PCs connected via a 10 Mbps Ethernet, the PC nodes can be kept synchronized within the deviation bound of 0.5 milliseconds. We plan to extend this study to one dealing with wireless network environments where GPS receivers cannot be used in most of the nodes.

**Acknowledgment:** The research reported here was supported in part by the NSF under Grant Numbers 99-75053 (NGS) and 00-86147 (ITR), and in part by the US DARPA (NEST Program) under Contract F33615-01-C-1902 monitored by AFRL. No part of this paper represents the views and opinions of any of the sponsors mentioned above.

## References

- [Ger94] Gergeleit, M., and Streich, H., "Implementing a Distributed High-Resolution Real-Time Clock using the CAN-Bus", *Proceedings of the 1<sup>st</sup> international CAN-Conference 94*, Sep. 94, Mainz.
- [IEE97] IEEE, "Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications", *IEEE Std. 802.11-1997*, Nov. 1997.
- [Kim97] Kim, K.H., "Object Structures for Real-Time Systems and Simulators", *IEEE Computer*, Vol. 30, No.8, Aug. 1997, pp. 62-70.
- [Kim99] Kim, K.H., Ishida, M., and Liu, J.Q., "An Efficient Middleware Architecture Supporting Time-Triggered Message-Triggered Objects and an NT-based Implementation", *Proc. 2nd IEEE CS Int'l Symp. on Object-Oriented Real-time Distributed Computing (ISORC '99)*, St. Malo, France, May, 1999, pp.54-63.
- [Kim00] Kim, K.H., "APIs for Real-Time Distributed Object Programming", *IEEE Computer*, June 2000, pp.72-80.
- [Kim01] Kim, K.H., Liu, J.Q., Miyazaki, H., and Shokri, E.H., "TMOES: A CORBA Service Middleware Enabling High-Level Real-Time Object Programming", *Proc.*

*ISADS 2001 (IEEE CS 5th Int'l Symp. on Autonomous Decentralized Systems)*, Dallas, TX, March 2001, p. 327-335.

[Kop97] Kopetz, H., "Real-Time Systems - Design Principles for Distributed Embedded Applications", *Kluwer Academic Publishers*, 1997, Chap. 3, page 45-70.

[Mil91] Mills, D.L. "Internet time synchronization: the Network Time Protocol", *IEEE Trans. Communications COM-39, 10*, Oct. 1991, p.1482-1493.

[NTP02] NTP.org, "Time Synchronization Server", <http://www.ntp.org>, 2002.

[Sol00] Solomon, D.A., Russinovich, M.E., "Inside Microsoft Windows 2000", 3rd Edition, *Microsoft Press*, 2000.

[Tur02] True Time, Inc., "Model GPS-PCI2 Data Sheet", [http://www.truetime.com/DOCSn/GPS\\_PCI\\_2.pdf](http://www.truetime.com/DOCSn/GPS_PCI_2.pdf), 2002.

# Proceedings

---

**Fifth IEEE International Symposium on  
Object-Oriented Real-Time Distributed Computing (ISORC 2002)**

---

**29 April — 1 May 2002 ♦ Washington, D.C.**

**Editors**

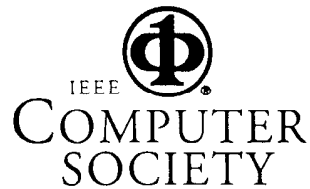
**Luiz Bacellar  
Peter Puschner  
Seongsoo Hong**

***Sponsored by***

IEEE Computer Society Technical Committee on Distributed Processing

***In cooperation with***

OMG and IFIP WG 10.4



Los Alamitos, California

Washington • Brussels • Tokyo

---