

# EECS 123:

## Introduction to Real-Time Distributed Programming

### Lecture : **Global Time & Clock Synchronization**

- Some materials in this slide-set are from Hermann Kopetz, 'Real-Time Systems - Design Principles for Distributed Embedded Applications', Kluwer Academic Publishers, 1997, Chap. 3, page 45-70.

Jan-07

UCI  
DREAM Lab



## Global Time & Clock Synchronization

- Time and Order
- Clock
  - Clock drift
  - Offset
  - Precision & Accuracy
- Global Time
  - Reasonable Condition
- Interval Measurement
- Overview of Clock Synchronization
- Clock Synchronization in TMOSM

Jan-07

UCI  
DREAM Lab



# Clock

- Digital (physical) clock in a computer .
  - Consists of a counter, and a physical oscillation mechanism that periodically generates a signal to increase the counter.
  - The periodic signal is called the **microtick** of the clock
  - The average duration between two consecutive microticks is the **granularity** of the clock.
- The **drift** of a physical clock  $k$  between microtick  $i$  and microtick  $i+1$   
$$:= \frac{\text{Duration (the instant of microtick } i, \text{ that of microtick } i+1)}{\text{Duration (true time claimed to be represented by microtick } i, \text{ that by microtick } i+1)}$$
- **Drift rate** during period  $P := | \text{drift\_during\_P} - 1 |$
- Real clocks have a varying drift rate that is influenced by environmental conditions.
  - For example, a change in the ambient temperature

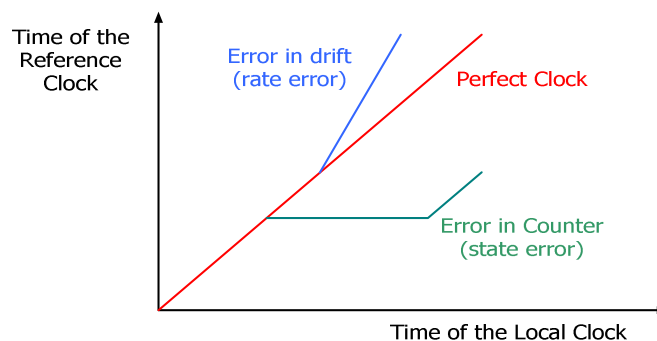
Jan-07

UCI  
DREAM Lab



# Failure Modes of a Clock

- The counter could be mutilated by a fault so that the counter value becomes erroneous (state error).
  - Easy to prevent by hardware manufacturers, e.g., use of ECC
- The drift rate of the clock could depart from the specified drift rate (rate error).



(Adapted  
from  
[Kop97])

Jan-07

UCI  
DREAM Lab



## Precision & Accuracy

- **Precision**
  - Given a set of clocks operating during the period of interest, the maximum offset/deviation of respective microticks of any two clocks is the **precision** of the clock-set.
- **Accuracy**
  - The maximum offset of a given clock from the external time reference (e.g., UTC) during the time interval of interest

Jan-07

UCI  
DREAM Lab



## Global Time

- Assume that a set of clocks are internally (within a distributed computing system) synchronized with a precision  $\Pi$  .  
I.E., for any two clocks  $j$  and  $k$ , the offset between any pair of corresponding microticks from  $j$  and  $k$ , respectively, is  $< \Pi$  .
- Select a subset of the microticks of each local clock  $k$  as a readable set of ticks.
  - If the readable set of ticks is the only set of ticks that can be read, clock  $k$  is a local implementation of the notion of **global time**.
  - Every member of the readable set of local microticks is called a **macrotick** (or a **tick**) of the global time.
- A **global time** in an environment equipped with a set of distributed physical clocks is thus an abstract notion that is **approximated** by proper selection of microticks generated from distributed physical clocks.

Jan-07

UCI  
DREAM Lab



## Reasonableness Condition

- Global time  $t$  is called **reasonable** if all local implementations of the  $t$  satisfy the condition

$$g > \Pi,$$

where  $g$  is the global time granularity and  $\Pi$  is the precision of the clock-set.

- To ensure that the synchronization error is **bounded** to less than one **macro-granule**, i.e., the duration between two ticks.
- Suppose this reasonableness condition is satisfied and a single event  $e$  can be detected in parallel by multiple distributed computing nodes using their own sensors.

The timestamps taken by distributed computing nodes using **their versions of the global time supported by their local clocks** for the event  $e$  can differ by at most one tick.

Jan-07

UCI  
DREAM Lab



## Reasonableness Condition (cont)

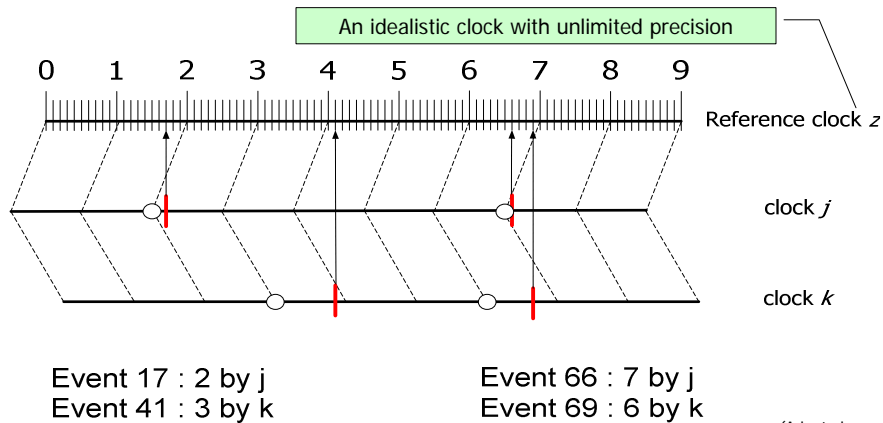
- It is not possible to reconstruct the temporal order of two events from the knowledge that their timestamps taken by two different nodes differ by one tick.
- If the timestamps of two events differ by two ticks, then the temporal order can be reconstructed because the sum of the synchronization and digitalization error is always less than 2 macro-granules.

Jan-07

UCI  
DREAM Lab



## Temporal order of two events with timestamps differing by one tick



(Adapted from [Kop97])

- Looks like a little crooked example

Jan-07

UCI  
DREAM Lab



## Interval Measurement

- An interval is delimited by two events, start event & terminating event.
- The synchronization error and the digitalization error can affect the measurement of these two events.
- The sum of these errors is less than  $2g$  because of the reasonableness condition, where  $g$  is the global time granularity.
- It follows that the true duration  $d_{true}$  of an interval is bounded by:

$$(d_{obs} - 2g) < d_{true} < (d_{obs} + 2g)$$

where  $d_{obs}$  is the difference between the start event observed by one node and the terminating event observed by another node.

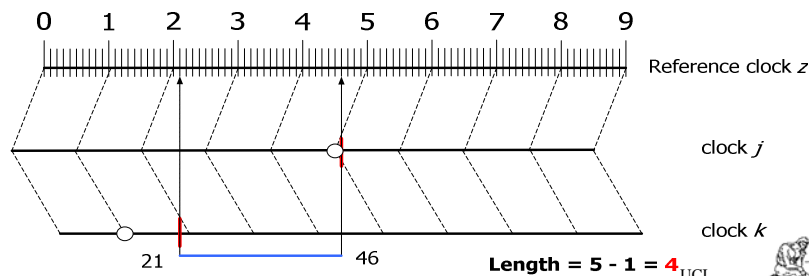
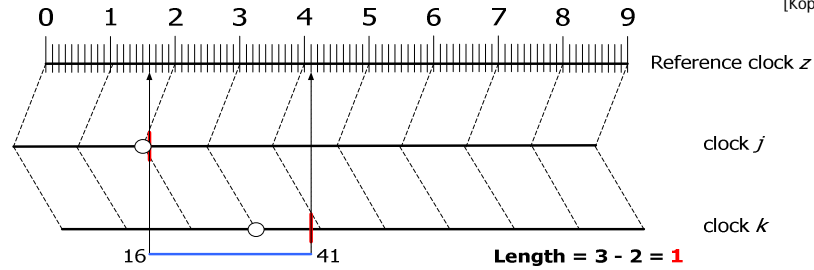
Jan-07

UCI  
DREAM Lab



## Errors in Interval Measurement

(Adapted from [Kop97])



Jan-07

UCI  
DREAM Lab



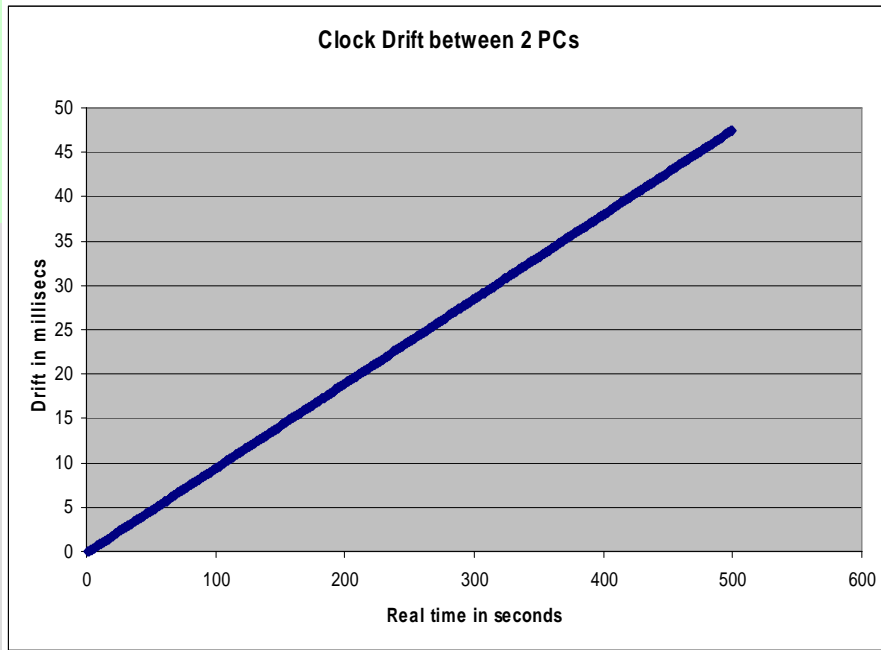
## Clock Synchronization

- A well-synchronized global time base is essential for proper and timely operation of distributed real-time systems.
- Local clocks in distributed computing nodes will diverge due to their difference in clock **drift** rates.
  - One experiment shows that a clock drift rate of a PC can be around  $10^{-4}$ .
- Each local clock should be adjusted **periodically** in order that, at any given time, the difference among local clocks of all participating distributed computing nodes may be **bounded within a specific deviation**.

Jan-07

UCI  
DREAM Lab





Jan-07

UCI  
DREAM Lab



## Clock Synchronization (cont)

- **Internal Clock Synchronization**
  - To ensure that the micro-ticks of all correct nodes occur within the specified precision  $\Delta$ , despite the varying drift rate of the local real-time clock of each node
  - [Centralized](#) vs. [Distributed](#) Synchronization Algorithm
  - [State correction](#) vs. [Rate correction](#)
- **External Clock Synchronization**
  - To keep a clock within a bounded accuracy with respect to an external source of standard time such as GPS

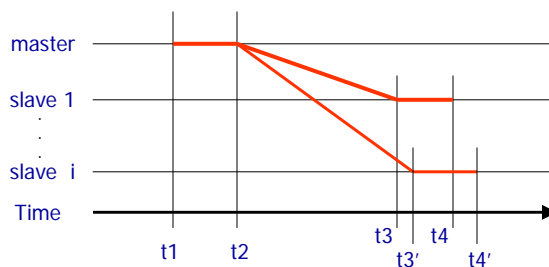
Jan-07

UCI  
DREAM Lab



## Simple Clock Synchronization Protocol

- **Master-slave** synchronization (centralized internal synchronization)
  - All local clocks will be synchronized with the local clock of the master node.
  - 1. The master takes a current timestamp based on its local clock ( $t_1$ ).
  - 2. The master broadcasts the timestamp as a clock synchronization message ( $t_2$ ).
  - 3. All slaves receive the clock synchronization message and create their local timestamps based on their local clocks ( $t_3, \dots, t_3'$ ).
  - 4. Slaves adjust their local clocks accordingly ( $t_4, \dots, t_4'$ ).

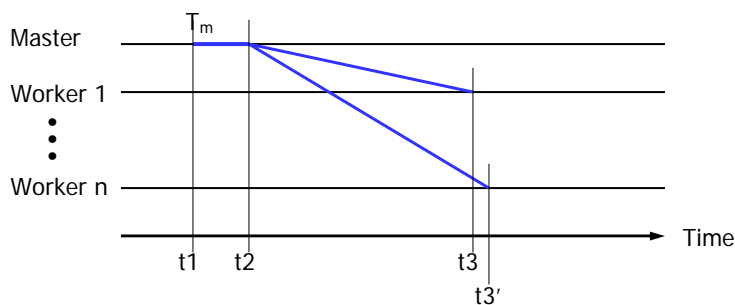


Jan-07

UCI  
DREAM Lab



## A Simple Master-Slave Synchronization



At  $t_1$ , the master node takes a timestamp ( $T_m$ ) and sends it to all worker nodes as a synchronization message.

At  $t_2$ , the packet containing a synchronization message leaves the local network interface card (NIC).

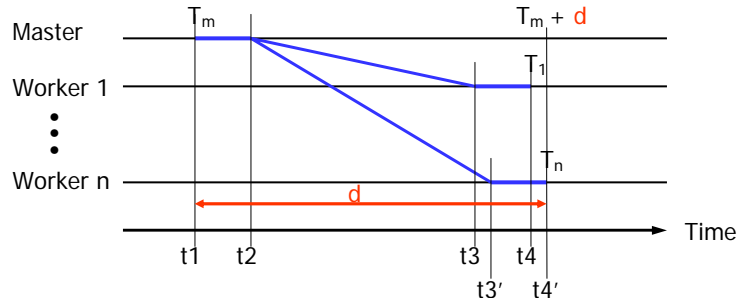
At  $t_3$ , and  $t_3'$ , the synchronization message arrives at NICs of each worker node.

Jan-07

UCI  
DREAM Lab



## A Simple Master-Slave Synchronization



At  $t_4$  and  $t_4'$ , each worker node reads its local clock and takes a timestamp ( $T_1$  and  $T_n$ ).

Suppose  $d$  is the amount of the time between  $t_4'$  and  $t_1$ , then by the time Worker  $n$  takes the timestamp  $T_n$ , the local clock message of the master node will reach at  $T_m + d$ .

Then Worker  $n$  can decide the amount of time its local clock drifts from that of the master node by calculating  $T_n - (T_m + d)$ .

→ This works only if Worker  $n$  knows the correct value of  $d$ .

Jan-07

UCI  
DREAM Lab



## A Simple Master-Slave Synchronization

Time spent during clock synchronization should be deterministic.

- $t_1 \sim t_2$  :
  - The process/thread responsible for taking and sending the timestamp in the master node **should not be preempted** during synchronization.
  - The highest priority should be to the process/thread responsible for clock synchronization
- $t_2 \sim t_3$  :
  - Communication delay should be **bounded** and **deterministic**.
  - **Isolate** the network environment from noisy sources to possible extents
  - Regulate accesses by distributed nodes to shared channels
- $t_3 \sim t_4$  :
  - The process/thread responsible for receiving the synchronization message and taking its local timestamp **should not be preempted** during synchronization.
  - The highest priority should be to the process/thread responsible for clock synchronization

Jan-07

UCI  
DREAM Lab



## Challenging Issues

---

- How to minimize or eliminate unpredictable interferences in the network during each synchronization round?
  - Isolate the network
  - Force a silent period during the clock synchronization
- How to create a non-preemptible on-alert condition in the responsible thread in each node during each synchronization round?
  - Let a thread/process with the highest priority handle the clock synchronization
  - Let a thread/process on worker nodes do busy-waiting for the synchronization message

Jan-07

UCI  
DREAM Lab



## TMOSM

---

- **TMOSM (TMO Support Middleware)** is the execution engine for TMOs
  - Middleware running on well established commercial software/hardware platforms, to support TMO programming model.
- TMOSM runs on among others Windows XP, Windows CE, and Linux platforms communicating with each other through Ethernet connection.
- TMOSM can support the execution of TMO-based applications with 10ms-level precision.

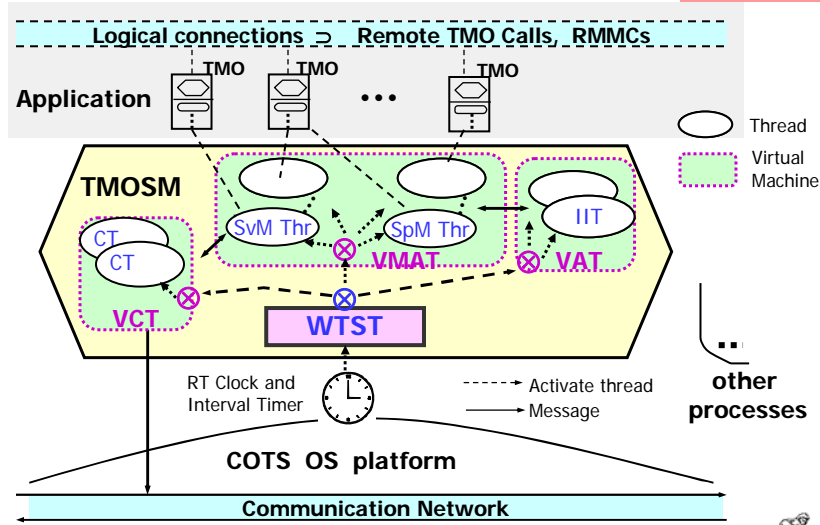
Jan-07

UCI  
DREAM Lab



# TMO Support Middleware on Windows XP & CE -- TMOSM / XP or CE / Socket

Currently running



Jan-07

UCI  
DREAM Lab



## Architecture of TMOSM

- Virtual machines (VMs) run **periodically**.
  - Easier analysis of the worst-case time behavior of the object execution engine without incurring any significant performance drawback.
- **WTST** (Watchdog Timer & Scheduler Thread) runs at the highest possible priority level, and manages the scheduling / activation of all other threads in TMOSM.
  - Activated **whenever thread switching needs to be performed**, e.g., upon expiration of a time-slice.
  - **Checks for any deadline violations** and if a violation is found, it provides an exception signal to the user.

Even those threads created by the node OS before TMOSM starts are executed only if WTST allocates some time-slices to them.

- **VCT** (VM for Comm Threads) manages the sending and receiving of **middleware messages**.

Jan-07

UCI  
DREAM Lab



## Architecture of TMOSM (cont)

- **VMAT** (VM for Main Application Threads) represents all application threads running TMO methods. Every time-slice conceptually belonging to the VMAT is allocated to a fairly selected application thread.
- **VAT** (VM for Auxiliary Threads) maintains a pool of threads serving as slaves to VMAT, mostly ordered by TMO method execution threads to execute the I/O functions.

Jan-07

UCI  
DREAM Lab



## Time in TMOSM

- TMOSM uses the time information from a **high-resolution performance counter** which the hardware provides.
- A **high-resolution timer** in Pentium III is a 64-bit register counter, which starts getting incremented when the machine is turned on and is incremented by one at every  $n$  clock cycles, where  $n$  is machine-dependent.
- When TMOSM starts, TMOSM reads a high-resolution performance counter via a Win32 API, `QueryPerformanceCounter()`, and sets this value as the **start time of DCSage**.
- The frequency of the performance counter can be retrieved via a Win32 API, `QueryPerformanceFrequency()`, which returns the counter number incremented in 1 sec.
- The time resolution TMOSM "**pretends**" to use is 1  $\mu$ sec.
- Current **DCSage** can be calculated by
$$\{ \text{Counter-Value (present)} - \text{Counter-Value (at the start time of DCSage)} \} * 1000000 / \text{Counter-Frequency} .$$

Jan-07

UCI  
DREAM Lab



## Clock Synchronization in TMOSM

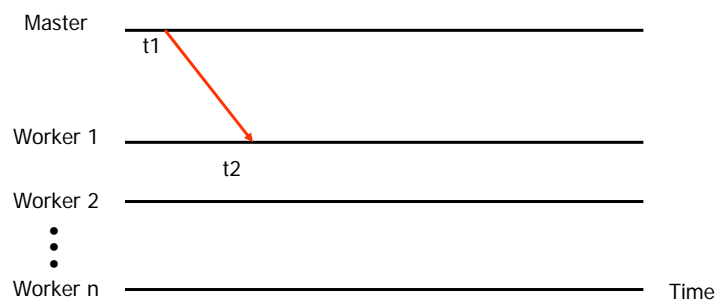
- Initial Clock Synchronization
  - Making all distributed nodes start keeping DCSage in a consistent manner during the initialization of their TMOSM instances
  - DCS Initialization
    - Clocks of slave nodes are synchronized when the master node broadcasts time for the first time.
  - Late Initialization
    - Late joining slave nodes request for special help from the master node
    - E.G., Resurrecting and rejoining a repaired node into a DCS running a TMO network.
- Clock Resynchronization
  - Making the difference among the DCSage's kept by distributed computing nodes be bounded within the specific range at all times

Jan-07

UCI  
DREAM Lab



## Initial Clock Synchronization



At the end of TMOSM initialization, all worker nodes are waiting for a synchronization message from the master node.

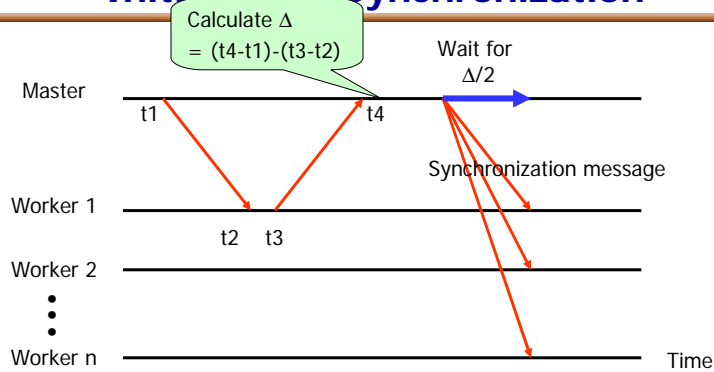
Assumption : Communication delays from the master node to worker nodes are likely to be a **constant** in a small area LAN environment **when the network is interference-free**.

Jan-07

UCI  
DREAM Lab



## Initial Clock Synchronization



The master node picks up one worker node as a *representative* worker node to calculate a round-trip communication delay ( $\Delta$ ).

A round-trip communication delay ( $\Delta$ ) can be estimated by exchanging messages between the master node and the representative worker node.

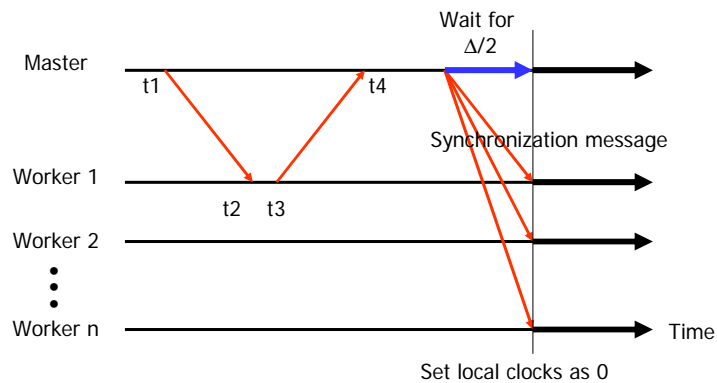
The master node broadcasts the synchronization message containing a one-way communication delay ( $\Delta/2$ ) to all worker nodes and waits for the duration equal to  $\Delta/2$ .

Jan-07

UCI  
DREAM Lab



## Initial Clock Synchronization



When worker nodes receive the synchronization message, set their local clocks as 0 and start normal execution.

The master node will set its local clock as 0 and starts normal execution after waiting for  $\Delta/2$ .

Jan-07

UCI  
DREAM Lab



## Initial Clock Synchronization

- Part I: During DCS initialization
  - All slave nodes are waiting for TMOSM\_STARTUP message.
  - The master node picks up one slave node as a **primary slave node** to compute the communication delay.
    - Communication delay between the master node and all slave nodes is **assumed** to be the same in the small-area LAN environment.
    - Round-trip communication delay ( $\Delta$ ) between the master node and the primary slave node is calculated.
  - The master node sends TMOSM\_STARTUP message containing master's system time (society time, i.e., C21age) and round-trip communication delay( $\Delta$ ) to all waiting slave nodes;
  - The master node then sleeps for a duration equal to  $\Delta/2$ ;
  - Then the master node sets the local DCSage to zero and starts execution;
  - As soon as a slave receives the TMOSM\_STARTUP message from the master node, it
    - adjusts its system time according to master's system time (master's system time +  $\Delta/2$ ).
    - Sets the local DCSage to zero and starts execution.

Jan-07

UCI  
DREAM Lab



## Initial Clock Synchronization (Cont.)

- Part II: During late initialization of a slave
  - This scheme is used to synchronize the clocks of the master and the slave node which is initialized after the DCS initialization;
  - The slave node sends a **registration** message to the master;
  - The master node responds by sending a TMOSM\_STARTUP message which contains the assigned node ID, master's system time, master's local DCS age, and **round-trip communication delay** ( $\Delta$ ) (**probably not accurate in this case**) ;
  - Upon receiving the TMOSM\_STARTUP message, the slave node sets the local DCS age to that of master's +  $\Delta/2$ , adjusts its system time according to master's system time (master's sys time +  $\Delta/2$ ) and starts execution.

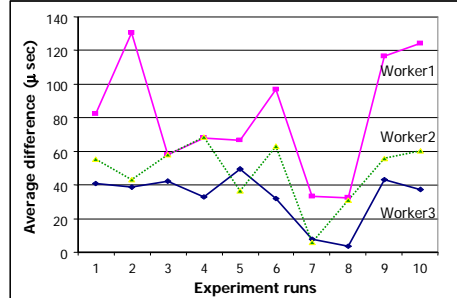
Jan-07

UCI  
DREAM Lab



## Results of Experiments with Initial Clock Synchronization

- 4 Pentium III + Windows 2000 machines connected via 100 Mbps Ethernet (Isolated)



- All local clocks in distributed nodes can be synchronized with the maximum deviation of **140 microseconds** after the initial clock synchronization.

1. After performing initial clock sync., the master node broadcasts its local timestamp to worker nodes every 1 millisecond for 10 milliseconds.
2. Every time each worker node receives the timestamp from the master, it takes its local timestamp
3. After receiving 10 timestamps from the master node, each worker node calculates the average difference between the master's clock and its local clock.  
(Network delay calculated during initial clock synchronization is reflected.)
4. Repeat step 1 through step 3 10 times.

Jan-07

UCI  
DREAM Lab



## Clock Resynchronization

- Use of WTST in Master to send clock resynchronization messages
  - The **thread** which sends the message must be **non-preemptible** and must be scheduled most frequently.
  - WTST has the highest priority (so cannot be preempted) and is invoked periodically every 3ms.
- Use of WTST in Slave to receive and process clock resynchronization messages
  - The receiving thread should pick the message as soon as it arrives.
  - The receiving thread must be **non-preemptible**.

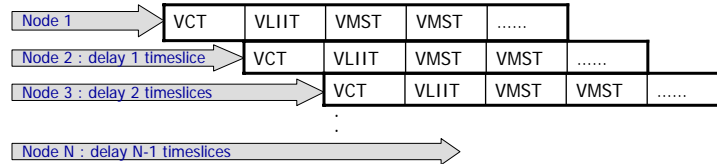
Jan-07

UCI  
DREAM Lab



## Regulation of the Message Communication

- For the regulation of the communication network :
  - The TMOSM allocates network bandwidth in a TDMA fashion among all of the participating nodes in the system.
  - The TMOSM instantiation in a computing node sends messages to the comm. network only via its VCT.
  - The execution times of VCTs in different nodes are staggered as follows



- All nodes use the same system-thread arrangement.
- After all the above regulation effects are done, the communication delay jitter is much smaller.
  - Experiment data (see the attached chart).

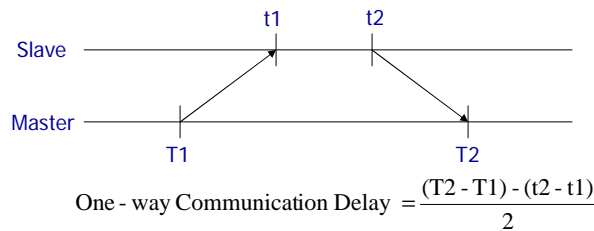
Jan-07

UCI  
DREAM Lab

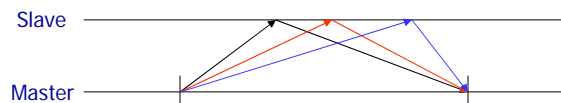


## Estimation of One-Way Comm Delay

- Calculating an expected one-way communication delay



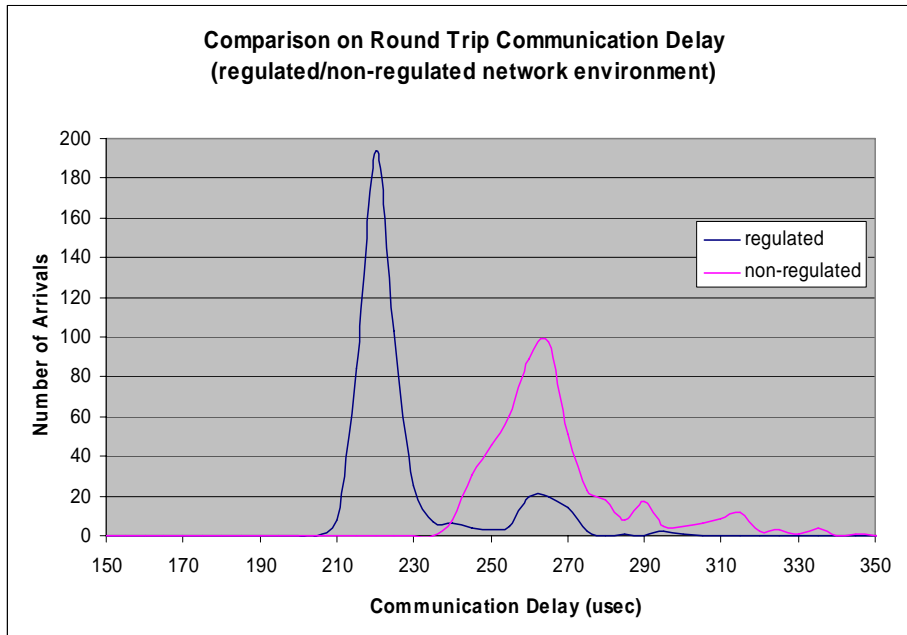
- The above approach has limitations.
  - These three round-trips will lead to the same one-way communication delay.



Jan-07

UCI  
DREAM Lab





Jan-07

UCI  
DREAM Lab



## Communication Delay

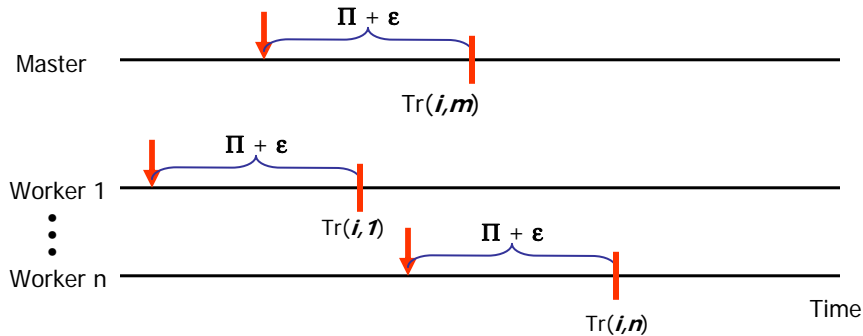
- According to the experiment on communication delay in a regulated network environment, 83% of the round-trip communication delay is bounded within the range from 210  $\mu$ S to 230  $\mu$ S.
- The communication delay between the master node and each slave node is **likely to be nearly constant** where clock synchronization messages are broadcasted in a small-area LAN environment.
- It may be ok to calculate only the communication delay between the master node and one selected slave node during TMOSM initialization phase, and use this value as an expected delay in transferring each clock synchronization message.

Jan-07

UCI  
DREAM Lab



## Clock Resynchronization - Force a Silent Period



- Clock resynchronization is initiated periodically based on the local clock.  
 $Tr(i,k)$  : the  $i$ -th clock resynchronization point at node  $k$

- Each node will enter the clock resynchronization mode when its local clock reaches at  $Tr(i,k) - \Pi - \epsilon$

$\Pi$  : Max deviation from the local clock of the master node

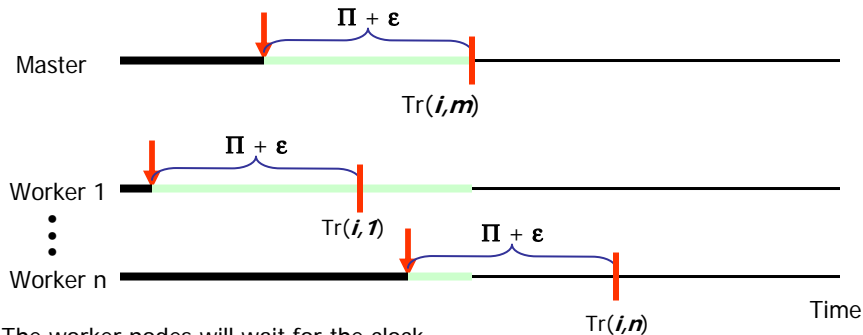
$\epsilon$  : Max length of the period in which a local network interface card can autonomously access the communication channel before becoming idle

Jan-07

UCI  
DREAM Lab



## Clock Resynchronization - Force a Silent Period



The worker nodes will wait for the clock resynchronization message from the master node.

When the master node enters the resynchronization mode, it will wait until the resynchronization point reaches based on its local clock.

By the time the master node reaches the resynchronization point, all participating distributed nodes are doing nothing but waiting for the resynchronization message from the master node

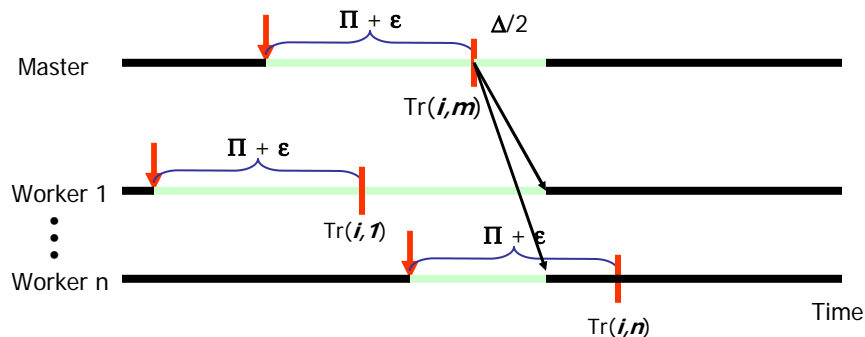
→ A silent period is imposed during the resynchronization mode !!

Jan-07

UCI  
DREAM Lab



## Clock Resynchronization - Force a Silent Period



Then, the master node broadcasts its local timestamp to worker nodes as the resynchronization message and waits for  $\Delta/2$ .

After waiting for  $\Delta/2$ , the master node aborts the resynchronization mode and starts the normal execution.

When each worker node receives the resynchronization message from the master node, it adjusts its local clock and starts normal execution.

Jan-07

UCI  
DREAM Lab



## Clock Resynchronization - Resynchronization Phase

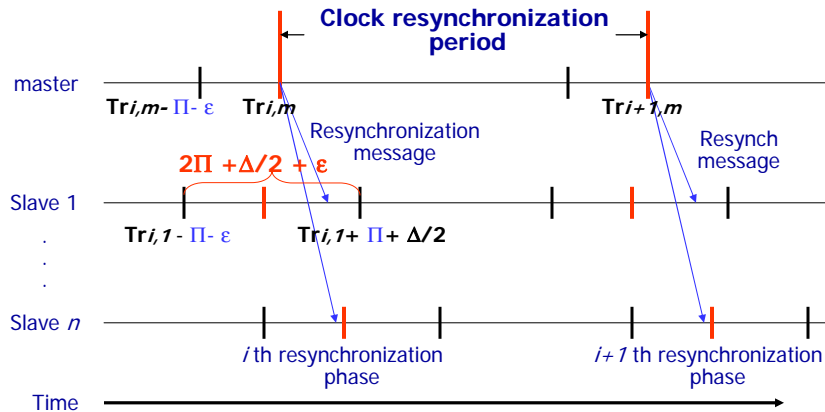
- Clock resynchronization is initiated periodically based on the local clock.
  - All participating nodes including both the master node and slave nodes will enter the [clock resynchronization mode](#) at  $T_{nr} - \Pi - \epsilon$ , where  $T_{nr}$  is the next clock resynchronization point,  $\Pi$  is the maximum deviation possible between the local clock of the master node and those of slave nodes, and  $\epsilon$  is the max length of the period in which a local NIC can autonomously access the comm channel (e.g., Ethernet) before becoming idle.
  - In the clock resynchronization mode, all threads except WTST executing the clock synchronization will be suspended.
  - The master node will wait until  $T_{nr}$  is reached according to its own clock, and then it will broadcast the local timestamp as a clock resynchronization message and exit from the clock resynchronization mode.
  - Slave nodes will wait until the arrival of the clock resynchronization message or [Resynchronization Time-Out](#) ( $2\Pi + \epsilon + \Delta/2$ ).

Jan-07

UCI  
DREAM Lab



## Clock Resynchronization - Resynchronization Phase



$T_{n,k}$  : time for  $n$ th clock resynchronization in Node  $k$   
 $\Pi$  : maximum deviation between local clocks  
 $\Delta$  : round-trip communication delay

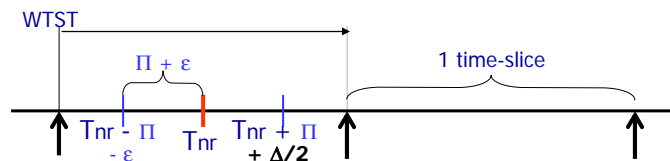
UCI  
DREAM Lab



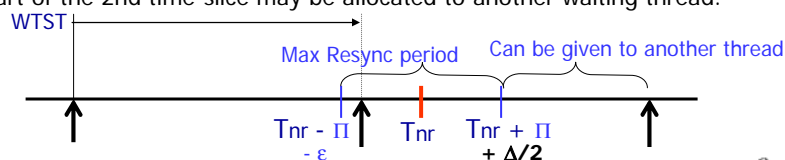
Jan-07

## Clock Resynchronization - Resynchronization Phase

- WTST of both the master node and slave nodes will check whether the  $T_{nr} - \Pi - \epsilon$  is within the next time-slice. If so, clock resynchronization routine will be executed.



- WTST of slave node whose local clock is ahead of that of the master node should wait to receive the clock resynch message by  $T_{nr} + \Pi + \Delta/2$ .
- The clock resynchronization routine may spend more than one time-slice but a part of the 2nd time-slice may be allocated to another waiting thread.



UCI  
DREAM Lab



Jan-07

## Clock Resynchronization - State Change

- If slave nodes receive the clock resynchronization message,
  - Take a local timestamp
  - Calculate the amount of difference between the master clock and its local clock (by reflecting one-way comm delay calculated during the initial clock synchronization), *clock\_diff*, then adjust the local clock.
    - If the local clock is slower than the master clock,
      - the local clock goes forward by *clock\_diff*.
    - If the local clock is faster than the master clock,
      - the local clock goes backward by *clock\_diff*.
      - TMOSM will be halted for a time equal to *clock\_diff* to prevent applications from experiencing the arrival of the same real-time instant twice.
  - If slave nodes reach the timeout ( $2\Pi + \Delta/2 + \epsilon$ ),
    - Faults have been detected !

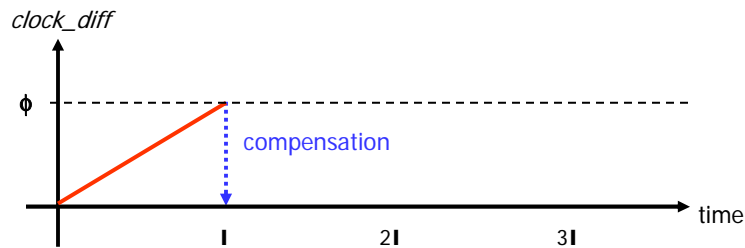
Jan-07

UCI  
DREAM Lab



## Clock Resynchronization - Adjust Local Clocks of Worker Nodes

- Suppose  $\phi$  is the amount of drift of the local clock of a given worker node from that of the master node after one clock resynchronization period ( $\Pi$ ),



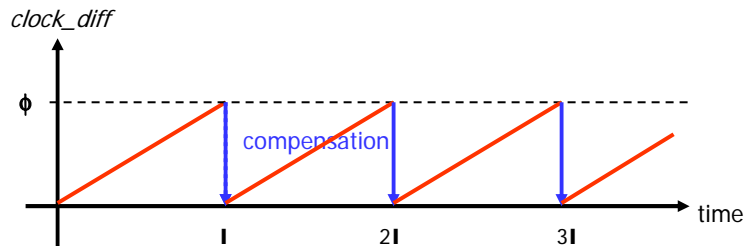
- Compensating amount is equal to *clock\_diff*.

Jan-07

UCI  
DREAM Lab



## Clock Resynchronization - Adjust Local Clocks of Worker Nodes



∴ If the clock drift rate is steady,  
a local clock can be kept within  $\phi$  from the master clock !!

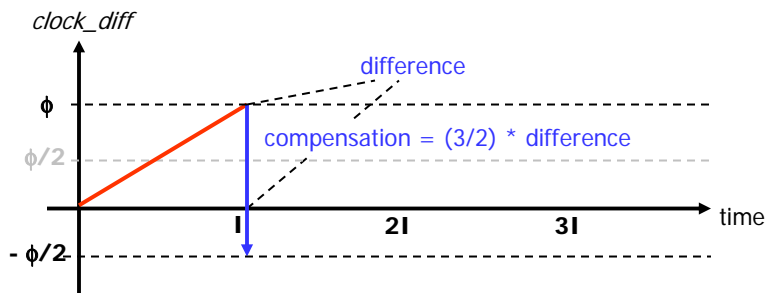
Jan-07

UCI  
DREAM Lab



## Clock Resynchronization - Optimization 1

- If the clock drift rate is steady, further reduced maximum deviation between the local clock of the master node and those of worker nodes could be achieved by **overcompensating the local clock at the first resynchronization round**.



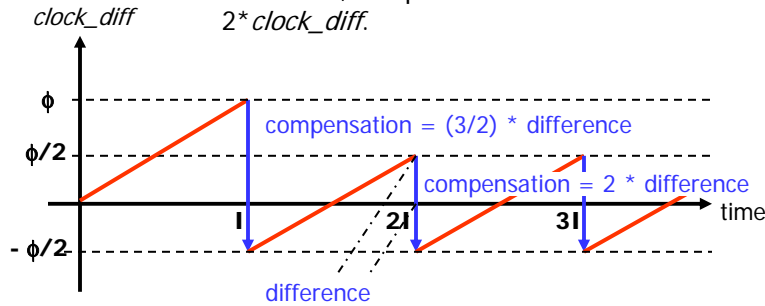
Jan-07

UCI  
DREAM Lab



## Clock Resynchronization - Optimization 1

- For the first clock resynchronization, compensate the local clock with  $(3/2) * clock\_diff$ .
- After then, compensate the local clock with  $2 * clock\_diff$ .



∴ A local clock can be kept within  $\phi/2$  from the master clock except **the first round !!**

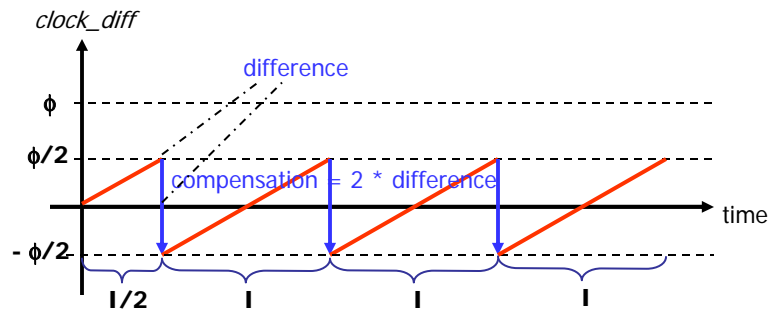
Jan-07

UCI  
DREAM Lab



## Clock Resynchronization - Optimization 2

- Further improvement can be achieved by
  - Performing the first clock resynchronization round after **only one half of the normal clock resynchronization interval ( $I/2$ )**.



∴ A local clock can be kept within  $\phi/2$  !!

Jan-07

UCI  
DREAM Lab



## Results of Experiments with Clock Resynchronization Using Optimization 1

- 4 Pentium III + Windows 2000 machines connected via 100 Mbps Ethernet (Isolated)

### Measurement

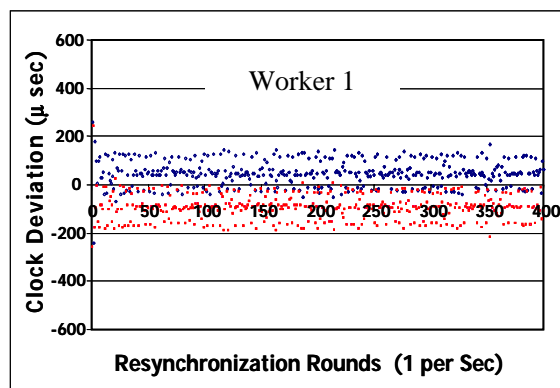
- When a synchronization message arrives at a slave,
  1. Calculate the difference between the master clock and the slave clock
- Just after adjustment of a local clock,
  1. The master node waits for messages from all slave nodes. A slave node sends a message to the master node as soon as clock resynchronization is finished.
  2. When the master node receives all message from slave nodes, send a local time stamp to all slave nodes.
  3. Each slave calculates the difference

Jan-07

UCI  
DREAM Lab



## Results of Experiments with Clock Resynchronization Using Optimization 1



- Blue dot : Master-Worker difference right before each clock resynchronization
- Red dot : Master-Worker difference right after each clock resynchronization

∴ The local clocks of all participating nodes can be kept within the maximum deviation of 500 microseconds.

Jan-07

UCI  
DREAM Lab



## Summary of Experiments

---

- Middleware-level clock synchronization can give 1ms-level precision in both initial clock synchronization and clock resynchronization.
  - Giving the highest priority to the thread responsible for clock synchronization
  - Regulating network environment
- The approach can work in fault-prone environments.
  - Failures of slaves do not complicate the procedure much.
  - Provisions must be made for handling failures of the master.
- To get higher precision in clock synchronization, a device-driver level approach is needed.

Jan-07

UCI  
DREAM Lab



## References

---

- [Kop97] Kopetz Hermann, '*Real-Time Systems - Design Principles for Distributed Embedded Applications*', Kluwer Academic Publishers, 1997, Chap. 3, page 45-70.
- [Kim02] Kim, K.H., Im, C.S., and Athreya, P., "Realization of a Distributed OS Component for Internal Clock Synchronization in a LAN Environment", *Proc. ISORC 2002 (5th IEEE CS Int'l Symp. on OO Real-time distributed Computing)*, Crystal City, VA, April 2002, pp.263-270. ([http://dream.eng.uci.edu/TMO/pdf/isorc2002\\_clock.pdf](http://dream.eng.uci.edu/TMO/pdf/isorc2002_clock.pdf))
- [MSDN1] "Platform SDK: Windows System Information About Time", [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sysinfo/time\\_94bp.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sysinfo/time_94bp.asp)
- [MSDN2] "Platform SDK: Windows Sockets API Reference", [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sysinfo/time\\_94bp.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sysinfo/time_94bp.asp)

Jan-07

UCI  
DREAM Lab

