Causality and Time

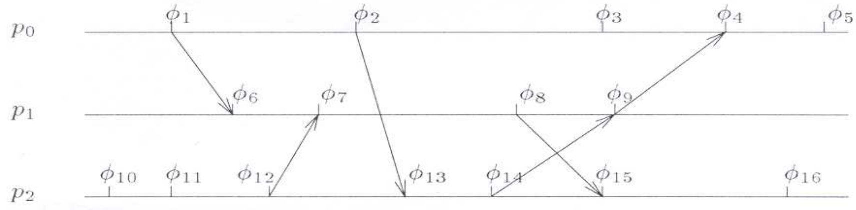
The Happens-Before Relation

- Because executions are sequences of events, they induce a total order on all the events.
- It is possible that two events by different processors do not influence each other, yet they are (arbitrarily) ordered by the execution.
  - The structure of causality between events is lost.

The happens-before relation

- Fix some execution α.
- Consider two events φ_1 and φ_2 by the same process p_j in α; φ_1 *casually influences* φ_2, if φ_1 occurs before φ_2.
- Consider two events φ_1 and φ_2 by different processes, p_i and p_j, respectively. Event φ_1 *causally influences* φ_2, if φ_1 is the event that sends message m from p_i to p_j, and φ_2 is the event in which the message m is received by p_j.
- Also, transitivity holds (if φ_1 causally influences φ_2 and φ_2 causally influences φ_3, then φ_1 causally influences φ_3).

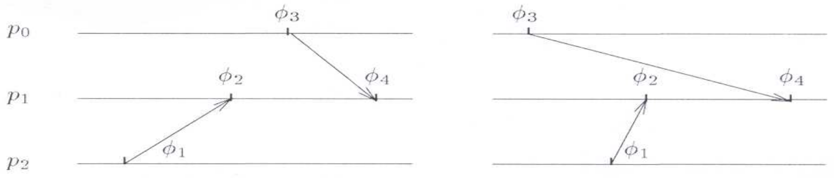
The Happens-Before Relation



Formally:

- Given two events φ_1 and φ_2 in α, φ_1 *happens before* φ_2, denoted $\varphi_1 \to \varphi_2$, if one of the following conditions hold:
  - φ_1, φ_2 are events by the same processor p_j and φ_1 occurs before φ_2 in α,
  - φ_1 is the send event of a message m from p_i to p_j, and φ_2 is the receive event of the message m by p_j,
  - There exists an event φ such that $\varphi_1 \to \varphi$ and $\varphi \to \varphi_2$.
- Obviously, → is an irreflexive partial order.

Causal Shuffles



(a) exec(C, σ)

(b) exec(C, π)

- The happens-before relation characterizes the causality relations in an execution.
  - If the events of an execution are reordered with respect to each other but without altering the happens-before relation, the result is still an execution and it is indistinguishable to the processors.
- **Definition**: Given an execution segment α = exec(C,σ), a permutation π of σ is a *causal shuffle* of σ if:
  - for all j, $0 \le j \le n-1$, $\sigma | j = \pi | j$, and
  - if a message m is sent during process' p_j event φ in α, then in π, φ precedes the delivery of m.

Causal Shuffles

Lemma 1

- Let $\alpha = exec(C,\sigma)$ be an execution fragment. Then, any total ordering of the events in σ that is consistent with the happens-before relationship of α, is a casual shuffle of σ.

Lemma 2

- Let $\alpha = exec(C,\sigma)$ be an execution fragment. Let π be a casual shuffle of σ. Then, $\alpha' = exec(C,\pi)$ is an execution fragment that is indistinguishable to α in all processes.

Logical Clocks

How can processes observe the happens-before relation in an execution α;

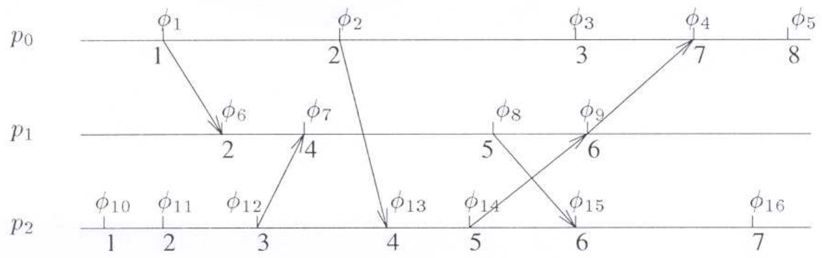
- With each event φ, we associate a *timestamp*, $LT(\varphi)$ (*Logical Time* of φ).
- We require an irreflexive partial order $<$ on the timestamps, such that for every pair of events, φ_1 and φ_2:

$$\text{if } \varphi_1 \rightarrow \varphi_2 \text{ then } LT(\varphi_1) < LT(\varphi_2)$$

Simple Algorithm to maintain logical timestamps correctly

- Each process p_j keeps a local variable LT_j, called its logical clock, which is a non-negative integer, initially 0.
- As part of each event φ, p_j increases LT_j to be one greater than the maximum of LT_j`s current value and the largest timestamp on any message received in this event.
- Every message sent by the event is time-stamped with the new value of LT_j.
- The *timestamp*, $LT(\varphi)$, associated with an event φ of process p_j, is the new value LT_j computed during the event.

Logical Clocks



- The partial order on timestamps is the ordinary < relation among integers.
- For each process p_j, LT_j is strictly increasing.

- **Theorem**: Let a be an execution, and let φ_1 and φ_2 be two events in a. If $\varphi_1 \rightarrow \varphi_2$, then $LT(\varphi_1) < LT(\varphi_2)$.

Vector Clocks

Logical Clocks - Negative Points

- **If $LT(\varphi_1) \geq LT(\varphi_2)$, then we know that φ_1 does not happen before φ_2.**

Is the converse true?

- **NO! It is possible that $LT(\varphi_1) < LT(\varphi_2)$ but it does not hold that $\varphi 1 \rightarrow \varphi 2$.**

- **The problem is that the happens-before relation is (in general) a partial order, whereas the logical timestamps are integers with the totally ordered <relation.**

Vector Clocks

Definition

- Two events φ_1 and φ_2 are *concurrent* in execution α, denoted $\varphi_1 \parallel \varphi_2$, if it does not hold neither $\varphi_1 \rightarrow \varphi_2$, nor that $\varphi_2 \rightarrow \varphi_1$.

- If $\varphi_1 \parallel \varphi_2$, then there are two executions α_1 and α_2, both indistinguishable from α, such that φ_1 precedes φ_2 in α_1, and φ_2 precedes φ_1 in α_2.

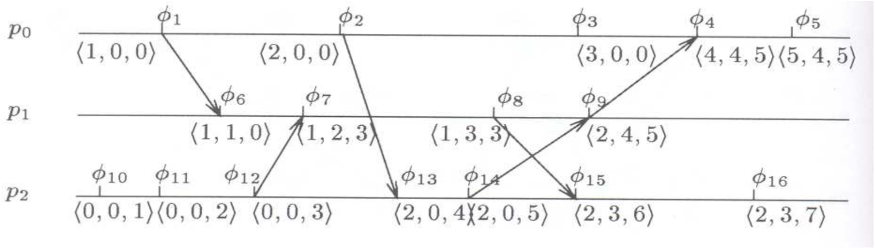
⇒ processes cannot tell whether φ_1 occurs before φ_2 or vice versa, and in fact, it makes no difference which order they occur.

Vector Clocks

- Each process p_j keeps a local n-element array $\mathbf{VC}_j$, called its *Vector Clock* (VC), each element of which is a non-negative integer, initially 0.
- As part of each event φ, p_j updates $\mathbf{VC}_j$ as follows:
  - $\mathbf{VC}_j[j]$ is incremented by 1
  - For each $i \neq j$, $\mathbf{VC}_j[i]$ is set equal to the maximum of its current value and the largest value for entry i among the timestamps of messages received in this event.
  - Every message, sent by φ, is time-stamped with the new value of $\mathbf{VC}_j$.

The *vector timestamp of φ*, **VC**(φ), is the value of $\mathbf{VC}_j$ at the end of φ.

Vector Clocks



Proposition: For each process p_j, in every reachable configuration, $\mathbf{VC}_j[j] \geq \mathbf{VC}_i[j]$, for all i, $0 \leq i \leq n-1$.

Partial Ordering of Vector Clocks

- Let $\mathbf{v}_1$ και $\mathbf{v}_2$ be two vectors of n integers. Then, *$v_1 \leq v_2$* if and only if, for every j, $0 \leq j \leq n-1$, $\mathbf{v}_1[j] \leq \mathbf{v}_2[j]$; *$v_1 < v_2$* if and only if $\mathbf{v}_1 \leq \mathbf{v}_2$ and $\mathbf{v}_1 \neq \mathbf{v}_2$.
- Vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ are *incomparable* if neither $\mathbf{v}_1 \leq \mathbf{v}_2$, nor $\mathbf{v}_2 \leq \mathbf{v}_1$.
- Vector timestamps are said to *capture concurrency* if for any pair of events φ_1 and φ_2, $\varphi_1 \parallel \varphi_2$ if and only if $\mathbf{VC}(\varphi_1)$ and $\mathbf{VC}(\varphi_2)$ are incomparable.

Vector Clocks

Theorem 1

Let α be an execution, and let φ_1 and φ_2 be two events in α. If $\varphi_1 \rightarrow \varphi_2$ then $\mathbf{VC}(\varphi_1) < \mathbf{VC}(\varphi_2)$.

Proof

Suppose that φ_1, φ_2 are events of the same process and let φ_1 precede φ_2. *Why is the claim true in this case;*

Suppose that φ_1 is the sending of a message m with vector timestamp T by p_i, and φ_2 is the receipt of the message by p_j. *Why is the claim true in this case;*

- *Does transitivity hold for the ≤ relation in vectors?*

Vector Clocks

Theorem 2

Let α be an execution, and let φ_1 and φ_2 be two events in α. If $\mathbf{VC}(\varphi_1) < \mathbf{VC}(\varphi_2)$ then $\varphi_1 \rightarrow \varphi_2$.

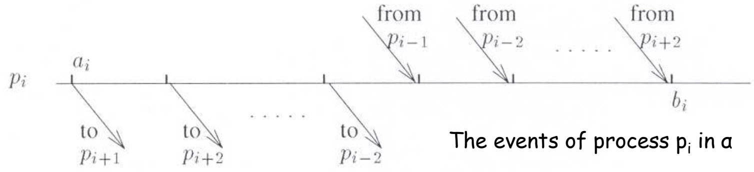
Proof

- Consider two concurrent events, φ_1 by p_i and φ_2 by p_j, $p_i \neq p_j$.
- Suppose $\mathbf{VC}_i[i](\varphi_1) = m$. The only way process p_j can obtain a value for the ith entry of its vector that is at least m is through a chain of messages originating at p_i (at event φ_1 or later).
- Such a chain would imply that φ_1 and φ_2 are not concurrent. A contradiction!
- Thus, $\mathbf{VC}_j[i](\varphi_2) < m \Rightarrow \mathbf{VC}_i(\varphi_1)$ cannot be smaller than $\mathbf{VC}_j(\varphi_2)$.

Similarly, the jth entry in $\mathbf{VC}_j(\varphi_2)$ cannot be smaller than the jth entry in $\mathbf{VC}_i(\varphi_1)$.

- Theorems 1 and 2 imply that $\varphi_1 \parallel \varphi_2$ if and only if $\mathbf{VC}(\varphi_1)$ and $\mathbf{VC}(\varphi_2)$ are incomparable.

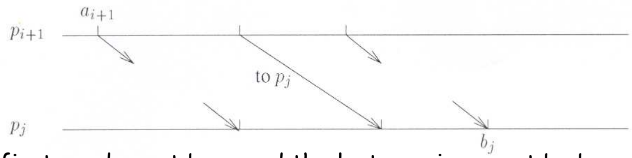
A Lower Bound on the Size of Vector Clocks



The events of process p_i in α

Consider a complete network and an execution α in which:

- Each process p_i sequentially sends a message to all processes other than p_{i-1}, i.e. it sends a message to $p_{i+1}, p_{i+2}, \ldots, p_{n-1}, p_0, p_1, \ldots, p_{i-2}$.
- After all the messages have been sent, each p_i sequentially receives all the messages sent to it in decreasing order of the sender's index, starting with pi-1 and wrapping around; p_i receives from $p_{i-1}, p_{i-2}, \ldots, p_0, p_{n-1}, p_{n-2}, \ldots, p_{i+2}$.



p_{i+1} and p_j in α

For each i, denote the first send event by a_i and the last receive event by b_i.

A Lower Bound on the Size of Vector Clocks

Lemma 3

- For each i, $0 \leq i \leq n-1$, $a_{i+1} \parallel b_i$ (where we let $a_n = a_0$).

Proof

- In α, a processor sends all its messages before it receives any message.
    - The causality relation is simple and does not include any transitively derived relations.
- No message is sent from p_{i+1} to p_i.

Lemma 4

- For each i and j, $0 \leq i \neq j \leq n-1$, $a_{i+1} \rightarrow b_j$.

Proof

Why is this true;

A Lower Bound on the Size of Vector Clocks

Theorem

If **VC** is a function that maps each event in α to a vector in R^k in a manner that captures concurrency, then $k \geq n$.

Proof

For each i, $0 \leq i \leq n-1$. By Lemma 3 ⇒ $a_{i+1} \parallel b_i$.

- Since **VC** captures concurrency ⇒ **VC**(a_{i+1}) and **VC**(b_i) are incomparable
- ⇒ there exists some coordinate r s.t. **VC**[r](b_i) < **VC**[r](a_{i+1}). Denote one of these indices by f(i).
- In this manner, we have defined a function

$$f: \{0, ..., n-1\} \rightarrow \{0, ..., k-1\}.$$

- We prove that $k \geq n$ by showing that f is 1-1.

A Lower Bound on the Size of Vector Clocks

- Assume, by the way of contradiction, that f is not 1-1.
  - $\exists$ two indices i and j, $i \neq j$, s.t. $f(i) = f(j) = r$.
- By the definition of f:

$$\mathbf{VC}[f(i)](b_i) < \mathbf{VC}[f(i)](a_{i+1}) \text{ and } \mathbf{VC}[f(j)](b_j) < \mathbf{VC}[f(j)](a_{j+1}) \Rightarrow$$

$$\mathbf{VC}[r](b_i) < \mathbf{VC}[r](a_{i+1}) \text{ and } \mathbf{VC}[r](b_j) < \mathbf{VC}[r](a_{j+1})$$

- By Lemma 4: $a_{i+1} \rightarrow b_j \Rightarrow \mathbf{VC}(a_{i+1}) < \mathbf{VC}(b_j)$.
- From all the above inequalities:

$$\mathbf{VC}[r](b_i) < \mathbf{VC}[r](a_{i+1}) \leq \mathbf{VC}[r](b_j) < \mathbf{VC}[r](a_{j+1}).$$

- A contradiction to Lemma 4!

Shared Memory Systems

The Happens-before relation

- Given two events φ_1 and φ_2 of an execution α, φ_1 *happens-before* φ_2, denoted by $\varphi_1 \rightarrow \varphi_2$, if one of the following conditions hold:
  - φ_1 and φ_2 are events of the same process, and φ_1 occurs before φ_2 in α,
  - φ_1 and φ_2 are conflicting events (both access the same shared variable and one of them is a write), and φ_1 precedes φ_2 in α,
  - there exists an event φ s.t. $\varphi_1 \rightarrow \varphi$ and $\varphi \rightarrow \varphi_2$.

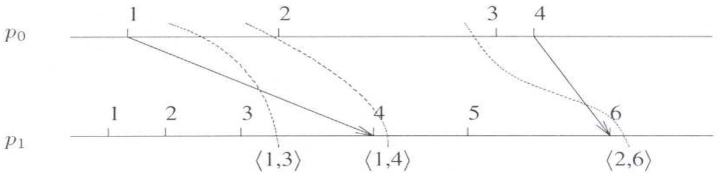
Examples of using Causality

Consistent Cuts

- No omniscient observer exists who can record an instantaneous snapshot of the system!
- Such a capability would be desirable:
  - restoring the system after a crash
  - determining whether there is a deadlock in the system
  - detecting termination
- **Simplifying Assumption**: At each event a process receives at most one message.
- Given an execution α, we number the steps of each process 1,2,3, ...
- A *cut* k through the execution is an n-vector $\mathbf{k} = <k_1, ..., k_{n-1}>$ of positive integers.
- Given a cut of an execution, one can construct a set of process states:
  - the state of process p_i is its state in α after the $\$\mathbf{k}_i\$$th computation event.

Examples of using Causality

Consistent Cuts



A cut k of an execution α is *consistent* if, for all i and j, the ($\mathbf{k}_i$+1)st event of p_i in α does not happen before the $\mathbf{k}_j$th event of p_j in α.

Finding the maximal consistent cut

- Given a cut **k** of an execution, find the "most recent" consistent cut that precedes (or at least does not follow) **k**.

Taking a Distributed Snapshot

- Instead of being given the upper bounding cut, processes are told when to start finding a consistent cut.

Modeling Physical Clocks

- Assume that there is a mechanism, called *hardware clock*, by which some time information is made available to the processes.

Formal definition of a timed execution

  - In each *timed execution*, associated with each process p_i, there is an increasing function HC_i from non-negative real numbers to non-negative real numbers. When p_i performs a step at real time t, the value of $HC_i(t)$ is available as part of the input of p_i's transition function.
  - p_i\`s transition function cannot change HC_i.
- We assume that $HC_i(t) = t + c_i$, where c_i is some constant offset.

Modeling Physical Clocks

- Executions are sequences of events that impose an arbitrary ordering on concurrent events.
- We can break apart an execution into n sequences, where each sequence represents the view of a processor.

Definition 1

- A *view with clock values* of a process p_i (in a model with hardware clocks) consists of an initial state of p_i, a sequence of events that occur at p_i, and a hardware clock value assigned to each event. The hardware clock values must be increasing, and if the sequence of events is infinite they must increase without bound.

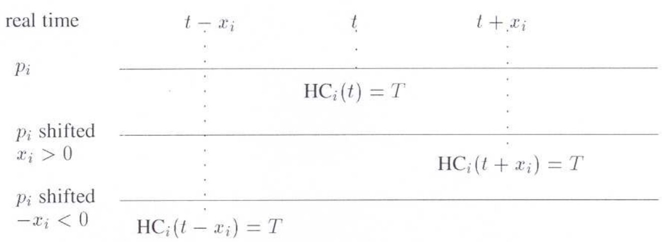
Definition 2

- A *timed view with clock values* of a process p_i (in a model with hardware clocks) is a view with clock values of p_i together with a real time assigned to each event. The assignment must be consistent with the hardware clock having the form $HC_i(t) = t + c_i$ for some constant c_i.

Timed Executions

- Given a timed execution α, time views with clock values can be extracted, denoted $\alpha \mid i$, for p_i's timed view with clock values.
- A set of n timed views $\{\eta_0, \ldots, \eta_{n-1}\}$, one for each process p_i, can be *merged* as follows:
  - The initial configuration is obtained by combining the initial states of all the timed views.
  - A sequence of events is obtained by interleaving the events in the timed views consistently with the real times.
  - Ties are broken by ordering all deliver events at time t before any computation events at time t, and breaking any remaining ties with processor indices.
- The result is denoted by $merge(\eta_0, \ldots, \eta_{n-1})$.
- Execution $merge(\eta_0, \ldots, \eta_{n-1})$ is admissible, if the timed views are *consistent*:
  - if a message is delivered to p_i from p_j at time t in η_i, but p_j does not send m to p_i before time t in η_j, $merge(\eta_0, \ldots, \eta_{n-1})$ is not a timed execution.

Shifting a Process in Time



- **Definition**: Let α be a timed execution with hardware clocks and let x be a vector of n real numbers. Define $shift(\alpha, x)$ to be $merge(\eta_0, \ldots, \eta_{n-1})$, where η_i is the timed view obtained by adding x_i to the real time associated with each event in $\alpha \mid i$.
- The result of shifting an execution is not necessarily an execution.
  - a message may not be in the appropriate processor's outbuf variable when a deliver event occurs.
- **Lemma**: Let α be a timed execution with hardware clocks HC_i, $0 \le i \le n-1$, and x be a vector of n real numbers. In $shift(\alpha,x)$:
  1. the hardware clock of p_i is $HC_i' = HC_i - x_i$, and
  2. every message from p_j to p_i has delay $\delta - x_i + x_j$, where δ is the delay of the message in α, $0 \le i, j \le n-1$.

The Clock-Synchronization Problem

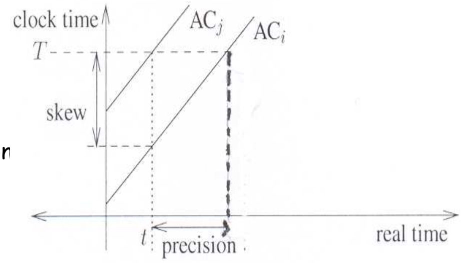
- Processes require to communicate to bring their clocks close together.
- Each process p_i has a special state component adj_i that it can manipulate.
- The adjusted clock of p_i is a function $AC_i(t) = HC_i(t) + adj_i(t)$.

Definition (achieving *ε-synchronized clocks*):

In every admissible timed execution, there exists a real time t_f s.t. the algorithm has terminated by real time t_f, and for all processes p_i and p_j, and all $t \geq t_f$, $|AC_i(t) - AC_j(t)| \leq \varepsilon$.
The value ε is called the *skew*.

Assumptions

- There exists non-negative constant d and u, $d \geq u$, s.t. in every admissible timed execution every message has delay within the interval [d-u, d].
- The value of u is the *uncertainty* in the message delay.

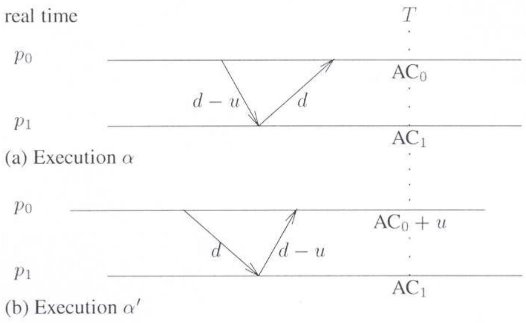


The Clock-Synchronization Problem – The Two Processes Case

A Simple Algorithm

- Process p_0 sets adj_0 to 0 and sends its current hardware clock value to process p_1.
- On receiving the message with value T, process p_1 sets $adj_1 = T + (d-u) - HC_1$ (so $AC_1 = T + (d-u)$)
- Best case: skew = 0.
- Worst case: skew = u.
- *What happens if adj_1 is adjusted to $T + d - HC_1$;*
- It is best to estimate the delay as d – u/2.
- *What is the skew of the algorithm then?*

The Two Processes Case



(a) Execution α

(b) Execution α'

Lemma: Te best skew that can be achieved in the worst case by a clock synchronization algorithm A for 2 processes p_0 and p_1 is u/2.

Proof

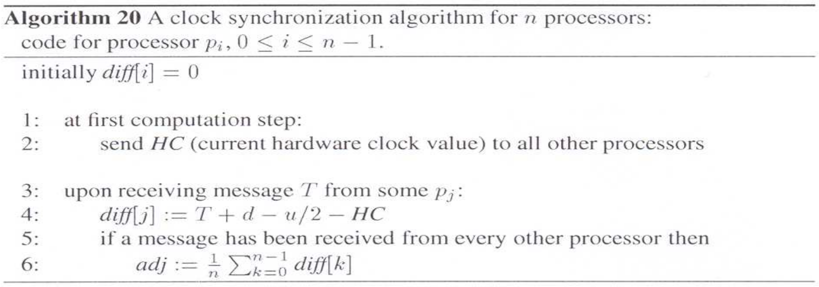
- Let α be any admissible execution in which the delay of messages from p_0 to p_1 is d-u and the delay of messages from p_1 to p_0 is d.
- Let AC_0 and AC_1 be the adjusted clocks at some time T after termination. Because A has skew ε, $AC_0 \geq AC_1 - \varepsilon$.
- Consider the execution α' = shift(α, <-u, 0>); α' is an admissible timed execution because all message delays are between d-u and d.
- At time T in α', the adjusted clock of p_0 is AC_0 + u, whereas the adjusted clock of p_1 is AC_1.
- Since A has skew ε, $AC_1 \geq (AC_0 + u) - \varepsilon$.
- Thus, $AC_0 \geq AC_0 + u - 2\varepsilon \Rightarrow \varepsilon \geq u/2$.

The Clock-Synchronization Problem – An Upper Bound

- Choose one of the processes as a center.
- Apply the two-processes algorithm between any process and the center.

➧ *What is the skew of this algorithm in the worst case;*

- We can do slightly better:



Algorithm 20 A clock synchronization algorithm for n processors: code for processor p_i, $0 \leq i \leq n-1$.

```
initially $diff[i] = 0$

at first computation step:
    send $HC$ (current hardware clock value) to all other processors

upon receiving message $T$ from some $p_j$:
    $diff[j] := T + d - u/2 - HC$
    if a message has been received from every other processor then
        $adj := \frac{1}{n}\sum_{k=0}^{n-1} diff[k]$
```

The Clock-Synchronization Problem – An Upper Bound

Theorem: Algorithm 20 achieves $u(1 - 1/n)$-synchronization for n processes.

Proof: Consider any admissible timed execution α of the algorithm.

- After p_i receives the message from p_j, $diff_i[j]$ holds p_i\`s approximation of the difference betweeen HC_j and HC_i. Because of the way $diff_i[j]$ is calculated, the error in the approximation is at most $+u/2$ or $-u/2$. Thus:

Lemma: For every time t after p_i sets $diff_i[j]$, $j \neq i$, it holds that $diff_i[j] = HC_j(t) - HC_i(t) + err_{ji}$, where err_{ji} is a constant with $-u/2 \leq err_{ji} \leq u/2$.

- By the definition of the adjusted clocks, at any time t after the algorithm terminates, the following holds:

$$|ACi(t) - ACj(t)| = | HC_i(t) + 1/n\ \Sigma_{k=0}^{n-1} diff_i[k] - HC_j(t) - 1/n\ \Sigma_{k=0}^{n-1} diff_j[k]| \quad (1)$$

- After some algebraic manipulation, we get:

$$1/n\ | HC_i(t) - HC_j(t) + diff_i[i] - diff_j[i] + HC_i(t) - HC_j(t) + diff_i[j] - diff_j[j] + \Sigma_{k=0,k\neq i,j}^{n-1} (HC_i(t) - HC_j(t) + diff_i[k] - diff_j[k])|$$

The Clock-Synchronization Problem – An Upper Bound

Proof (continued)

By laws of absolute value and the fact that diffi[i] = diffj[j] = 0, this expression is

$$(1) \leq 1/n(|HC_j(t) - HC_i(t) + diff_j[i] |^{(*)} + |HC_i(t) - HC_j(t) + diff_i[j]|^{(\sim)} + \Sigma_{k=0,k\neq i,j}^{n-1} | HC_i(t) - HC_j(t) + diff_i[k] - diff_j[k]|^{(\#)})$$

$^{(*)}$ -> difference between p_i\`s knowledge of its own clock and $p_j$\`s estimate of p_i\`s clock: $|err_{ij}| \leq u/2$

$^{(\sim)}$ -> the difference between p_j\`s knowledge of its own clock and $p_i$\`s estimate of p_j's clock: $| err_{ji} | \leq u/2$

$^{(\#)}$ -> difference between p_i\`s estimate of $p_k$'s cloxk and $p_j$\`s estimate of p_k's clock:

$$| HC_i(t) - HC_j(t) + HCk(t) - HC_i(t) +err_{ki} - HCk(t) + HC_j(t) -err_{kj}| \leq u.$$

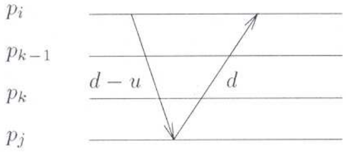
$\Rightarrow$ So, the entire expression $\leq 1/n(u/2 + u/2 + (n-2) u) = 1/n(n-1)u = (1 - 1/n)u$, as needed!

The Clock-Synchronization Problem – A Lower Bound

Theorem: For every algorithm that achieves ε-synchronized clocks, ε is at least u(1 – 1/n).

Proof: Consider any clock synchronization algorithm *A* that achieves ε-synchronized clocks.

- Let α be an admissible timed execution of *A* with hardware clocks HC_i s.t. for each process p_i and p_j, i < j:
  - the dealy of every message from p_i to p_j is exactly d-u,
  - the dealy of every message from p_j to p_i is exactly d.
- Let AC_i be the adjusted clock of p_i in α after termination, $0 \le i \le n-1$.



A Lower Bound

Lemma: For each k, $1 \le k \le n-1$, $AC_{k-1}(t) \le AC_k(t) - u + \varepsilon$.

Proof: Pick any k, $1 \le k \le n-1$.

- Define α' = shift(α, **x**), where $x_i = -u$ if $0 \le i \le k-1$ and $x_i = 0$ if $k \le i \le n-1$.
- Consider two processes p_i and p_j with i < j.
- If $j \le k-1$ or $k \le i$, then the delays from p_i to p_j are d-u and the delays from p_j to p_i are d.
- Otherwise ($i \le k-1 < j$), the delays from p_i to p_j are d and the delays from p_j to p_i are d-u.
- Thus, α' is admissible and *A* much achieve ε-synchronized clocks in α.
- Since processes are shifted earlier in real time, t is also after termination in α' $\Rightarrow$ $AC'_{k-1}(t) \le AC'_k(t) + \varepsilon$.
- We have $AC'_{k-1}(t) = AC_{k-1}(t) + u$ and $AC'_k(t) = AC_k(t)$.
- Putting all these pieces together, gives:

$$AC_{k-1}(t) \le AC_k(t) - u + \varepsilon, \text{ as needed.}$$

The Clock-Synchronization Problem – A Lower Bound

Proof of Theorem (continued):

- Since A achieves ε-synchronized clocks, it holds that:

$$AC_{n-1}(t) \le AC_0(t).$$

- We apply the lemma repeatedly to finish the proof:

$$\begin{aligned}
AC_{n-1}(t) &\le AC_0(t) + \varepsilon \\
&\le AC_1(t) - u + \varepsilon + \varepsilon = AC_1(t) - u + 2\varepsilon \\
&\le AC_2(t) - u + \varepsilon - u + 2\varepsilon = AC_2(t) - 2u + 3\varepsilon \\
&\le \ldots \\
&\ldots \\
&\le AC_{n-1}(t) - (n-1)u + n\varepsilon
\end{aligned}$$

$\Rightarrow (n-1)u \le n\varepsilon$

$\Rightarrow \varepsilon \ge (1 - 1/n)u$, as needed!