Clock Synchronization

Pedro Ferreira do Souto

Departamento de Engenharia Informática
Faculdade de Engenharia
Universidade do Porto

Outline

Outline

Outline

Outline

Outline

Outline

Outline

Outline

Outline

Outline

Model Overview

- Processes have access to local read-only physical clocks, which are subject to a very small rate of drift.
  - A process' local time is obtained by adding the value of the physical clock to the value of a local "correction" variable.
- The communication network is fully connected, so that every process can send a message directly to every other process.
- Processes are able to broadcast a message to all the processes at the same time.
- All messages are delivered within a fixed amount of time plus or minus some uncertainty.

Protocol Outline

- The protocol runs in rounds, resynchronizing every so often to correct for the clock drift.
- At every round, each process obtains a value for each of the other processes' clocks, and sets its clock to an average of those values computed with a fault-tolerant averaging function.
- This function is designed to be immune to some fixed maximum number, f, of faults.
- From a practical point-of-view, two capabilities are important:
  1. The reintegration of repaired processes so that they can synchronize their clocks
  2. The establishing of a synchronization among the clocks
     - ⋆ The algorithm can be seen as a way to maintain the clocks synchronized. But, how do they get synchronized? I.e., how is the initial synchronization performed in the face of clock drift, uncertainty in the message delivery time, and arbitrary process faults?

Outline

Model: Processes (1/2)

This is a paper before I/O Automata, but the main ideas are there.

- Processes communicate by sending messages to each other.
- Each process has a physical clock that is not under its control.
- Processes are interrupt-driven. Events are modeled as messages. E.g.:
  - A START message might signal initial system start-up.
  - A TIMER message might signal that the physical clock has reached a designated time.
- The processing time of an arriving message is considered instantaneous.
- A process is an automaton with a set of states and a transition function.

Model: Processes (2/2)

- A transition function describes:
  1. The new state the process enters;
  2. The messages it sends;
  3. The timers it sets.

  as a function of
  1. The process' current state;
  2. The messages it received;
  3. The physical clock time.
- At a process step the process:
  1. Receives a message;
  2. Changes state;
  3. Sends out some messages.
- If a process is **nonfaulty**, the new state and the messages sent obey the transition function. Otherwise the process is **faulty**.
  - Arbitrary (or *Byzantine*) faults can be modelled by not restricting the state changes or messages sent by faulty processes.

Model: Clocks

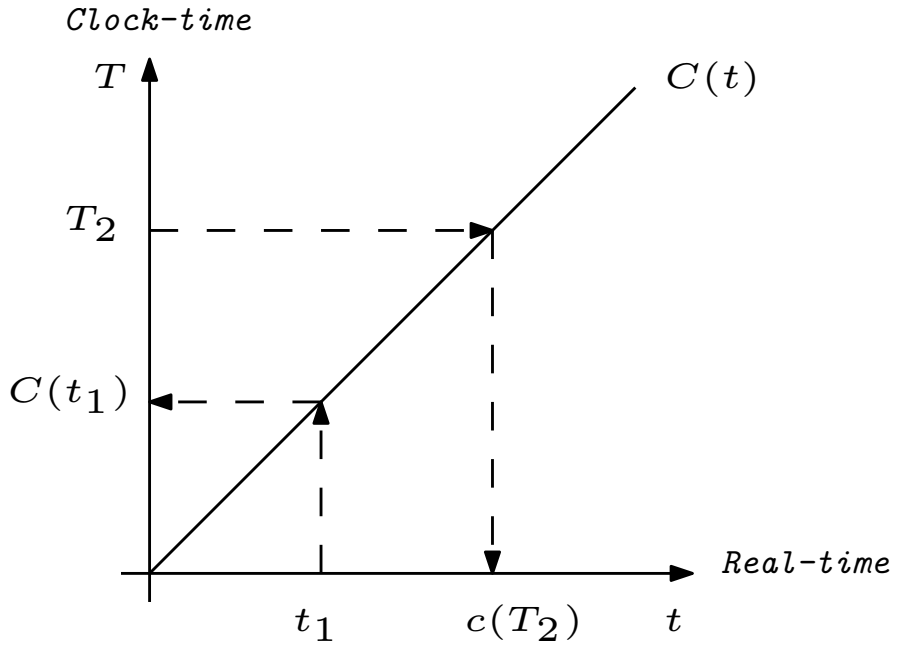
- A **clock** is a monotonicallly increasing, everywhere differentiable function from $\mathbb{R}$ to $\mathbb{R}$, interpreted as being a function from real times to clock times, or vice-versa.

Clock-Time vs. Real-time

- Lower-case letters denote real times;
- Upper-case letters denote clock times.

Likewise:

- A clock from real time to clock times is denoted with upper case
- And its inverse is denoted by the same name in lower case.



Sytem (Definition) Consists of a set of processes and a set of clocks, the **physical clocks**, from real times to clock times, one clock per process.

- The physical clock for process p will be denoted Ph_p.

Model: Communication system

- Each process can communicate directly with all processes, including itself.
- The communication system is modeled by a global message buffer.
  - When a process sends a message at real time t to another process, the message is placed in the message buffer together with a time t' greater than t.
  - At real time t', the message is received by the proper recipient and is deleted from the buffer..
  - The value $t' - t$ is the **message delay**.
- In its initial state, the message buffer contains no messages except for START messages, exactly one for each process, together with their scheduled delivery times.
- When a process p sets a timer, say for time T, a TIMER message with recipient p and delivery time $Ph_p^{-1}(T)$ is placed in the message buffer, provided that the latter is greater than the current real time. If it is not, no message is placed in the buffer.

Outline

ρ-Bounded Clocks

ρ-bounded Clock (Def) A clock C is $\rho - bounded$ provided that for all t

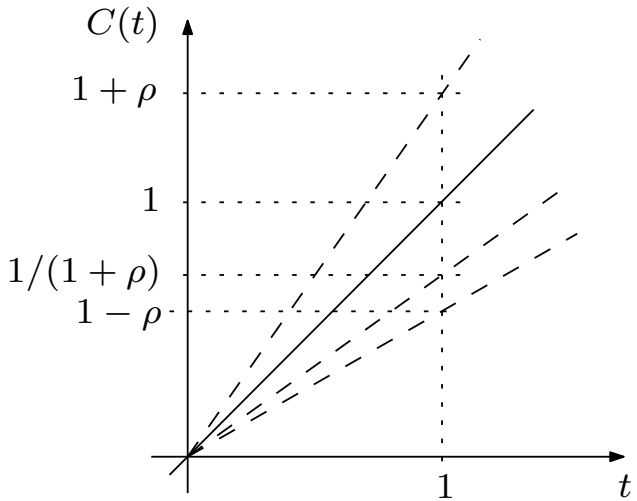
$$1/(1+\rho) \leq dC(t)/dt \leq 1+\rho$$

where $\rho > 0$ is a very small constant.

- I.e. a ρ-bounded clock is clock whose drift is small: the amount of clock time elapsed in a *rho*-bounded clock during some real time interval is close to the amount of real time that has elapsed.

Corollary (Because $0 < \rho < 1$.)

$$1-\rho \leq dC(t)/dt \leq 1/(1-\rho)$$



Note For a small ρ the lower bounds $1/(1+\rho)$ and $1-\rho$ are very close. Similarly the upper bounds.

ρ-bounded (Real time) Clock (Def) A clock c is $\rho - bounded$ provided that for all T

$$1/(1+\rho) \leq dc(T)/dT \leq 1+\rho$$

ρ-Bounded Clocks Properties (1/2)

Note By definition, the inverse of a ρ-bounded clock is itself a ρ-bounded clock.

Lemma 1 Let C be any $(\rho - \textit{bounded})$ clock. If $t_1 \leq t_2$, then

$$(t_2 - t_1)/(1+\rho) \leq C(t_2) - C(t_1) \leq (1+\rho)(t_2 - t_1)$$

Proof: Straightforward by the mean value theorem.

Lemma 2 Let C and D be clocks. Then for any t_1 and t_2:

$$\text{(a) } |(C(t_2) - t_2) - (C(t_1) - t_1)| \leq \rho|t_2 - t_1|$$
$$\text{(b) } |(C(t_2) - D(t_2)) - (C(t_1) - D(t_1))| \leq 2\rho|t_2 - t_1|$$

Proof: (a) Note that
$|(C(t_2) - t_2) - (C(t_1) - t_1)| = |(C(t_2) - C(t_1)) - (t_2 - t_1)|$.
Suppose $t_2 \geq t_1$ and $C(t_2) - C(t_1) \geq t_2 - t_1$. (One of 4 cases) Then:

$$\begin{aligned} |(C(t_2) - C(t_1)) - (t_2 - t_1)| &= (C(t_2) - C(t_1)) - (t_2 - t_1) \\ &\leq (1+\rho)(t_2 - t_1) - (t_2 - t_1), \text{by Lemma 1} \\ &= \rho|t_2 - t_1| \end{aligned}$$

ρ-Bounded Clocks Properties (2/2)

Lemma 2 **Proof** of part (b)

$$
\begin{aligned}
&|(C(t_2) - D(t_2)) - (C(t_1) - D(t_1))| \\
&\quad = |((C(t_2) - t_2) - (C(t_1) - t_1)) - ((D(t_2)) - t_2) - (D(t_1)) - t_1))| \\
&\quad \leq |(C(t_2) - t_2) - (C(t_1) - t_1))| + |(D(t_2)) - t_2) - (D(t_1)) - t_1)| \\
&\quad \leq 2\rho|t_2 - t_1|, \text{by part (a).}
\end{aligned}
$$

Lemma 3 Let C and D be clocks, and $T_1 \leq T_2$.
Assume that $|c(T) - d(T)| \leq \alpha$ for all $T, T_1 \leq T \leq T_2$.
Let $t_1 = min\{c(T_1), d(T_1)\}$ and $t_2 = max\{c(T_2), d(T_2)\}$.
Then, for all $t, t_1 \leq t \leq t_2$,

$$|C(t) - D(t)| \leq (1 + \rho)\alpha$$

Proof: By case analysis on the values of the inverse clocks:

1. $c(T_1) \leq t \leq c(T_2)$
2. $d(T_1) \leq t \leq d(T2)$
3. $c(T_2) < t < d(T_1)$
4. $d(T_2) < t < c(T_1)$

Outline

Problem Definition: Assumptions (1/2)

Each process p has a local variable $CORR$, which provides a correction to its physical clock to yield the local time.

- During an execution, p's local variable $CORR$ takes on different values.
- Let $CORR_p(t)$ be a function that returns the value of p's local variable $CORR$ at real time t.
- For a particular execution, we define the **local time** for p to be function $L_p = Ph_p + CORR_p$.
- A **logical clock** of p is Ph_p plus the value of $CORR_p$ at some time.
  - Let C_p^0 denote the **initial logical clock** of p, given by Ph_p plus the value, in p's initial state, of $CORR_p$. Let c_p^0 denote the inverse function of C_p^0
- Each time p adjusts $CORR_p$, it can be thought of as changing to a new logical clock.
- The local time can be thought of as a piecewise continuous function, each of whose pieces is part of a logical clock.

Problem Definition: Assumptions (2/2)

(A1) All clocks are ρ-bounded, including those of faulty processes, for some small constant ρ.

- Since faulty processes are permitted to take arbitrary steps, faulty clocks would not increase their power to affect the behavior of nonfaulty processes.

(A2) There are at most f faulty processes, for a fixed constant f, and the total number of processes in the system, n, is at least $3f + 1$.

(A3) The delay for every message is in the range of $[\delta - \epsilon, \delta + \epsilon]$, for nonnegative constants δ and ϵ, with $\delta > \epsilon$.

(A4) A START message arrives at each process at time T^0 on its logical clock C_p^0. Furthermore, for all nonfaulty processes p and q, $|c_p^0(T^O) - c_q^0(T^0)| \leq \beta$

- I.e. all the nonfaulty processes wake up within an interval of length β when their logical clocks reach T^0.

We denote the real time $c_p^0(T^0)$ by t_p^0.

We let $tmax^0 = max\{t_p^0, \text{such that p is nonfaulty}\}$, and analogously for $tmin^0$. These are respectively the latest and earliest real times when START messages arrive at nonfaulty processes.

Problem Definition: Properties

γ-agreement: For all $t \geq tmin^0$ and all non-faulty p and q:

$$|L_p(t) - L_q(t)| \leq \gamma$$

- I.e., at any real time, all the nonfaulty process's (logical) clocks differ by at most γ.

$(\alpha_1, \alpha_2, \alpha_3)$-validity: For all $t \geq t_p^0$ and all nonfaulty p:

$$\alpha_1(t - tmax^0) - \alpha_3 \leq L_p(t) - T^0 \leq \alpha_2(t - tmin^0) + \alpha_3$$

- I.e. the local time of a nonfaulty process increases in some linear relation to real time.
- When α_1 and α_2 are close to 1, and α_3 close to 0, then $L_p(t) - T^0$ is close to $t - t_p^0$. I.e. the amount of elapsed clock time is close to the amount of elapsed real time.

Outline

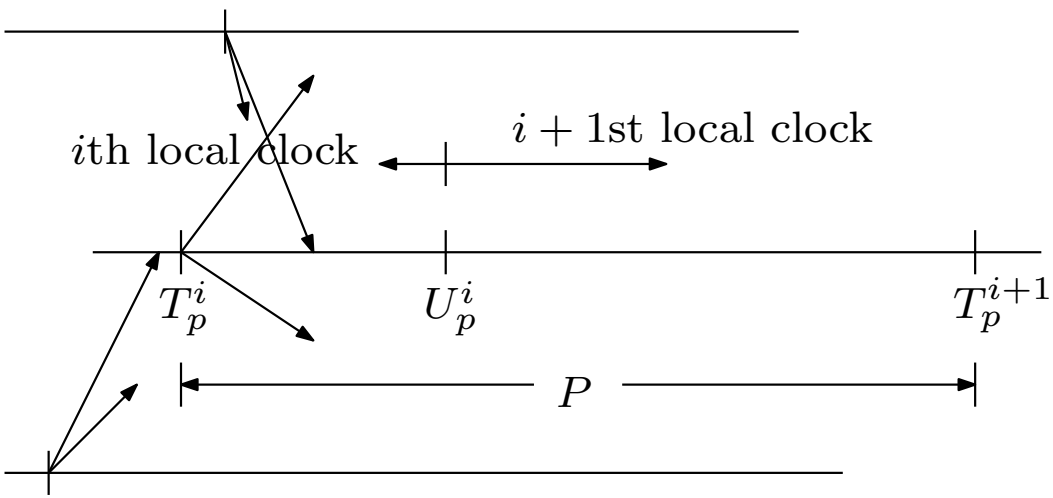
Outline

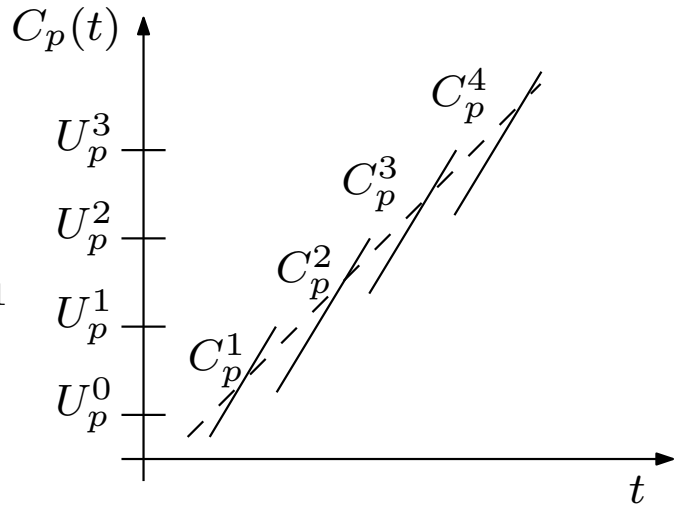
Algorithm: Outline (1/2)

- The algorithm is round based.
- In each round resynchronization occurs at a fixed local time.
- The ith round for process p is triggered by its ith logical clock reaching some value T^i.
- When p's ith logical clock reaches T^i, p broadcasts a T^i message.
- Meanwhile, p collects T^i messages from as many processes as it can, within a particular bounded amount of time, measured on its logical clock.
- The bounded amount of time is of length $(1+\rho)(\beta+\delta+\epsilon)$, and is chosen to be just large enough to ensure that p receives T^i messages from all the nonfaulty processes.
- After waiting for this amount of time, p averages the arrival times of all the T^i messages received, using a particular fault-tolerant averaging function.

Algorithm: Outline (2/2)

- The resulting average is used to calculate an adjustment to p's correction variable, $CORR$ thereby switching p to its $(i+1)$st logical clock.
- The process p then waits until its $(i+1)$st logical clock reaches time $T^{i+1} = T^i + P$, by setting a timer, and repeats the procedure. P is the length of a round in local time.





Algorithm: Averaging Function

- The fault-tolerant averaging function is derived from those used in the paper on approximate agreement that will be presented later.
- The function is designed to be immune to some fixed maximum number, f, of faults.
  1. It throws the f highest and the f lowest values.
  2. It then applies some ordinary averaging function to the remaining values.
  3. In the paper, the authors have chosen the midpoint of the range of the remaing values, which causes the error to be halved at each round.
- It is possible for the clock to be set backwards in this algorithm. However, there are known techniques for stretching a negative adjustment out over the resynchronization interval.

Outline

TIOA For Clock Synchronization: Signature

```
automaton ClockSyncWL(n, f: Nat, ρ, β, δ, ϵ, P: Real,
                      i: Nat)
  signature
    input     start(t: Real)
    input  receive(m: Real, j: Nat, const i: Nat) where j ≠ i
    output    broadcast(m: Real, const i: Nat)
    internal  update
```

- ClockSyncWL is the specification for a single process in the protocol, each of which has a non-negative integer identifier
- n, f, ρ, β, δ, ϵ, P are the protocol parameters as defined above
- We model the broadcast channel by $n \times n$ bounded channels
  - For each process i there is a channel to process j
  - Each channel from process i to process j has a broadcast(m,i) input (instead of a send(m,i,j) input)
  - By TIOA composition, a single broadcast(m,i) action by the ClockSyncWL automaton will put message m in each of the channels connecting process i to the remaining processes

TIOA For Clock Synchronization: States

```
states
  T: discrete AugmentedReal := ∞
  U: discrete Real := 0
  CORR: discrete Real := 0
  ARR: Array[Nat, discrete Real] := constant(0)
  Ph: Real :=
derived variables
  Local := Ph + CORR
```

`T` Local time for beginning of next round

`U` Local time for beginning of next logical clock

`CORR` Correction value for computing local time from local physical clock

`ARR` Arrival times array: one element per process with the local arrival time of the most recent message received from that process.

`Ph` Local physical clock

`Local` Local clock time, this is derived from the state variables

TIOA For Clock Synchronization: Initial States

The protocol assumes that the local clocks are initially synchronized

- Because we assume that CORR is initially 0, this imposes some constraints on the values of Ph of all processes
- The **initially** clause can only express a predicate that relates the automaton's initial state and parameters
- A workaround might be to include the initial values of the different physical clocks as a parameter of the automaton

TIOA For Clock Synchronization: Transitions (1/2)

input start(T^0)
  eff
    $T := T^0$

- The protocol assumes that at T^0, the local clocks are synchronized
- We assume that T^0 is not in the past, i.e. $T^0 \geq$ Local
- These assumption and the start() input ensure that all correct processes receive the messages broadcast by correct processes in the first round
- An alternative would be to do away with the start() action and initialize T with T^0
  - But we would need also to assume explicitly that no correct process misses any message broadcast by correct processes in the first round

TIOA For Clock Synchronization: Transitions (2/2)

```
input receive(m, j, i)
  eff
    ARR[j] := Local

output broadcast(m, i)
  pre
    m = T ∧ Local = T ∧ Local > U
  eff
    U := Local + (1+ρ)(β + δ + ε)
```

```
internal update
  local AV, ADJ: discrete Real
  pre
    Local = U ∧ Local > T
  eff
    AV := mid(reduce(ARR))
    ADJ := T + δ − AV
    CORR := CORR + ADJ
    T := T + P
```

- AV and ADJ are local auxiliary variables, they are not part of the state

Procedures

mid(M): applied to a multiset M of real numbers, returns the midpoint of the set of values in the multiset. (The midpoint is the arithmetic mean of the smallest and the largest elements in the multiset.)

reduce(A): applied to an array A, returns the multiset consisting of the elements of the array, with the f largest and the f smallest elements removed.

TIOA For Clock Synchronization: Trajectories

trajectories
  stop when
    $(\mathrm{Local} = \mathrm{T} \wedge \mathrm{Local} > \mathrm{U}) \vee (\mathrm{Local} = \mathrm{U} \wedge \mathrm{Local} > \mathrm{T})$
  evolve
    $1/(1+\rho) \leq \mathbf{d}(\mathrm{Ph}) \leq 1 + \rho$

- Until the start transition, the physical clock is allowed to increase in an unconstrained way
- The occurrence of start forces the process to execute a broadcast action
- After that, the broadcast and update transitions will be enabled alternately
  - It might be more explicit to add a variable named e.g. phase (which might be a derived variable)
  - The last term in the the preconditions for broadcast and update ensures that each of them is executed only once when the first

Outline

Notation (1/2)

- Let $T^i = T^0 + iP$ and $U^i = T^i + (1+\rho)(\beta + \delta + \epsilon)$
- For each $i \in \mathbb{N}_0$, every process p broadcasts T_i when its ith logical clock C_p^i reaches time T^i, i.e. at real time t_p^i.
- When its ith logical clock reaches U^i, i.e. at real time u_p^i, process p resets its $CORR$ variable, thereby switching to a new logical clock, denoted C_p^{i+1}.
- The process moves through an infinite sequence of clocks: $C_p^0, C_p^1, \ldots$ where C_p^0 is in force in the interval of real time $(-\infty, u_p^0)$, and each C_p^i, $i \geq 1$, is in force in the interval of realtime $[u_p^{i-1}, u_p^i)$
- The (real-time) interval $[t_p^i, t_p^{i+1})$ constitutes round i for process p
- Let $tmin^i = min\{t_p^i : \text{ for all p nonfaulty}\}$, and similarly for $tmax^i$, $umin^i$ and $umax^i$.

Notation (2/2)

- For p and q nonfaulty, let $ARR_p^i(q)$ denote the time of arrival, measured on p's logical clock C_p^i, of a T^i message from q to p, sent when q's ith logical clock C_q^i reaches time T^i
  - The authors claim, and prove, that C_p^i is in force by the time this message arrives.
- Let ARR_p^i denote the multiset of value $ARR_p^i(q)$ for all q.
- Let AV_p^i denote the value of AV calculated by p using the values in ARR_p^i, i.e. the value computed by throwing out the f earliest and the f latest values and taking the midpoint of the remainder
- Let ADJ_p^i denote the corresponding value of ADJ calculated by p, i.e., the "adjustment" calculated by subtracting the average from the "expected" arrival time $(T^i + \delta)$. Thus, $Cp^{i+1} = C_p^i + ADJ_p^i$

Parameters (1/2)

- Parameter ρ (drift rate), δ (media message delay) and ϵ (uncertainty in the delay) are determined by the hardware and low-level communication protocols employed.
- The parameters P (round length) and β (how closely in real time processes reach the same bound) are design parameters
- The smaller β more closely synchronized the clocks will be
- However, the smaller β the smaller P must be, i.e. the more often processes will need to synchronize.
- However, P cannot be arbitrarily small. In order for the algorithm to work correctly P must be sufficiently large to ensure that:
  1. After a nonfaulty process p resets its clock, the local time at which p schedules its next broadcast is greater than the local time on the new clock, at the moment of reset.
  2. The message sent by a nonfaulty process p at round i, which will be used to set the $i+1$st logical clock, arrives at a nonfaulty process q **after** q sets its ith logical clock (otherwise it will overwrite its previously sent value and be used to set p's ith logical clock.)

Parameters (2/2)

- It can be shown that sufficient conditions relating the parameters are:

$$P > 2(1+\rho)(\beta+\epsilon) + (1+\rho)max\{\delta, \beta+\epsilon\} + \rho\delta \quad (1)$$

$$P \leq \beta/(4\rho) - \epsilon/\rho - \rho(\beta+\delta+\epsilon) - 2\beta - \delta - 2\epsilon \quad (2)$$

  It follows that:

$$\beta \geq 4\epsilon + 4\rho(4\beta+\delta+4\epsilon+max\{\delta, \beta+\epsilon\}) + 4\rho^2(3\beta+2\delta+3\epsilon+max\{\delta, \beta+\epsilon\})$$

- If P is fixed, then β is roughly $4\epsilon + 4\rho P$.
  - This value can be obtained by solving the upper bound on P, (2) above, for *beta* and neglecting terms of order ρ or higher.

Theorem 4

Let p and q be nonfaulty processes and $i \geq 0$

(1) If $i \geq 1$, then $|ADJ_p^{i-1}| \leq (1+\rho)(\beta+\epsilon)+\rho\delta$

(2) If $i \geq 1$, then $U^{i-1} + ADJ_p^i < T^i$

(3) $|t_p^i - t_q^i| \leq \beta$

(4) If $i \geq 1$, then $t_p^i + \delta - \epsilon > u_q^{i-1}$

1. The adjustment ADJ_p^{i-1} used for p's ith logical clock is bounded.
2. The time to broadcast i messages is still in the future when p's ith logical clock is started.
3. p begins round i within β real time of any other nonfaulty process
4. p's round i messages arrives at q after q has already set its ith logical clock.

- Note that (2) and (4) are the conditions that impose a lower bound on the value of P.
- The proof is by induction on i.

γ-Agreement

Theorem 16

The algorithm guarantees γ-agreement, where

$$\gamma = \beta + \epsilon + \rho(7\beta + 3\delta + 7\epsilon) + 8\rho^2(\beta + \delta + \epsilon) + 4\rho^3(\beta + \delta + \epsilon)$$

Proof Must show that $|L_p(t) - L_q(t)| \leq \gamma$, for all nonfaulty p and q, and all $t \geq tmin^0$.

- Case analysis, depending on whether or not p and q use logical clocks with the same index. Result must hold in both cases.

- In practice $\rho < 1E^{-6}$, thus γ is determined mostly by β and ϵ. Essentially, if we need a synchronization within ms (μs, ns), β and ϵ need to be of the same order of magnitude (or smaller).
  - As stated above $\beta \geq 4\epsilon + 4\rho P$, thus even if the second term is negligible wrt ϵ, β is bounded by ϵ, i.e. the uncertainty of the communication channel delay

$(\alpha_1, \alpha_2, \alpha_3)$-Validity

Theorem 19

The algorithm preserves $(\alpha_1, \alpha_2, \alpha_3)$-validity, where

$$\alpha_1 = 1 - \rho - \epsilon/\lambda, \qquad \alpha_2 = 1 + \rho + \epsilon/\lambda, \qquad \alpha_3 = \epsilon$$

where $\lambda = (P - (1+\rho)(\beta+\epsilon) - \rho\delta)/(1+\rho)$.

Note: λ is the length of the shortest round in real time. (This is at least P minus the maximum adjustment.)

Proof: Must show for all, $t \geq t_p^0$ and all nonfaulty p that

$$\alpha_1(t - tmax^0) - \alpha 3 \leq L_p(t) - T^0 \leq \alpha_2(t - tmin^0) + \alpha_3$$

- Proof relies on two lemmas, which are prooved by induction.

- To keep the clocks close to real time, we want $\alpha_1 \approx 1$, and similarly for α_2. Because, $\rho << 1$, we have $\lambda \approx P - (\beta + \epsilon)$. Thus we'd like $P >> \epsilon$. However, as we have seen, increasing P increases β.

Outline

Reintegration of a Repaired Process (1/3)

- Let p be the process to be reintegrated into the system.
- During some round i, p will gather messages from the other processes and perform the same averaging procedure as that described previoulsy to obtain a value for its correction variable such that its clock becomes synchronized.
- Since p's clock is now synchronized, it will reach T^{i+1} within β of every other nonfaulty process.
- At that point p is no longer faulty and rejoins the main algorithm, sending out T^{i+1} messages.

Reintegration of a Repaired Process (2/3)

- Note that p can recover at an arbitrary time during an execution.
- As soon as it recovers, it begins collecting T^i messages for all plausible values of T^i.
- It is necessary that p identify an appropriate round i at which it is able to obtain all the T^i messages from nonfaulty processes.
- Since p might recover during the middle of a round, p must first observe the arriving messages, allowing part of a round to pass before it begins to collect messages.
- After p has determined that it should use T^i messages to update its clock, it continues to collect T^i messages.
- It must wait a certain length of time, as measured on its clock, in order to guarantee that it has received T^i messages from all nonfaulty processes. Immediately after p determines it was waited long enough, it carries out the averaging procedure and determines a value for its correction variable.

Reintegration of a Repaired Process (3/3)

Claim: p reaches T^{i+1} on its new clock within β of every other nonfaulty process.

1. It does not matter that p's clock begins initially unsynchronized with all the other clocks: the arbitrary value will be compensated for in the subtraction of the average arrival time.
2. It does not matter that p is not broadcasting a T^i message; p is being counted as one of the faulty processes, which could always fail to send a message. Furthermore, processes do not treat themselves specially, so it does not matter that p fails to receive a message from itself.
3. It does not matter that p adjusts its correction variable whenever it is ready (rather that at the time specified for correct processes). The adjustment is only the addition of a constant so the (additive) effect of the change is the same in either case.

Outline

Outline

Establishing Synchronization: Overview

- The algorithm establishes synchronization of clocks that initially may have arbitrary values.
- It handles Byzantine failures of the processes, uncertainty in the message delivery time and clock drift.
- To achieve this it relies on a combination of:
  - elapsed physical time, and
  - one additional message type

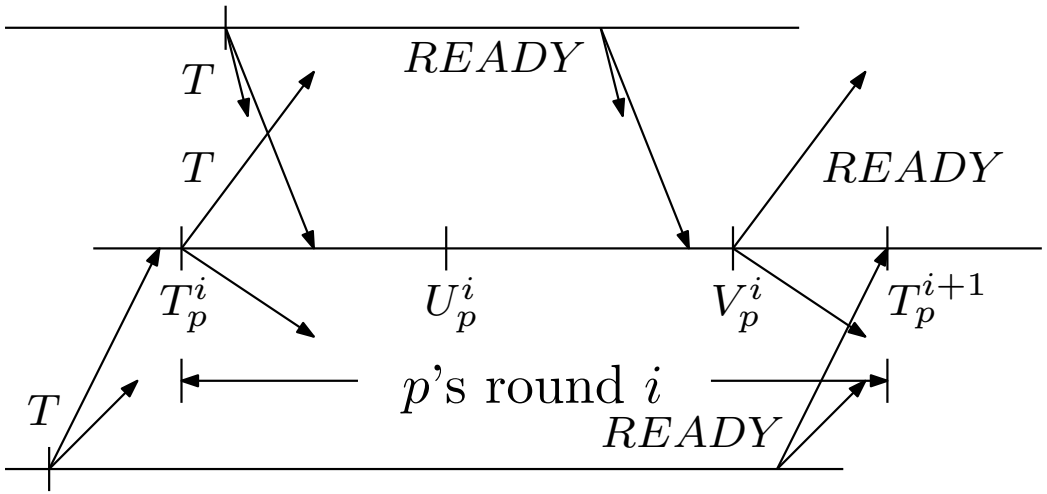
  rather than on the local/logical times.
- The algorithm runs in rounds.
- During each round, the processes exchange clock values and use the same fault-tolerant averaging function as before to calculate the corrections to their clocks.
- Each round contains an additional phase in which the processes exchange messages to signal that they are ready to begin the next round.
- The algorithm guarantees that nonfaulty processes begin each round within $\delta + 3\epsilon$ of each other.

Establishing Synchronization: Informal Description (1/2)

- At the beginning of each round, each nonfaulty process p broadcasts its local time.
- Then p waits for an interval of length $(1+\rho)(2\delta+4\epsilon)$, which is long enough for p to receive a similar message from each nonfaulty process.
  - The algorithm guarantees that nonfaulty processes begin each round within $\delta+3\epsilon$ of each other.
- At the end of this interval, p calculates the adjustment that it will make to its clock at the current round,
  - But the adjustment is made only at the end of the round
- Then p waits a second time interval of length $(1+\rho)(4\epsilon+4\rho(\delta+2\epsilon)+2\rho^2(\delta+4\epsilon))$ before sending out additional messages,
  - This ensures that these messages are not received while the other nonfaulty processes are in their first waiting intervals.

Establishing Synchronization: Informal Description (2/2)

- At the end of its second waiting interval, p broadcasts a *READY* message indicating that it is ready to begin the next round.
  - However, if p receives $f + 1$ *READY* messages during its second waiting interval, it terminates its second interval early, and goes ahead and broadcasts *READY*.
- As soon as p receives $n - f$ *READY* messages,
  - it updates the clock according to the adjustment calculated earlier,
  - and begins its next round by broadcasting its new clock value.



Outline

TIOA For Clock Synchronization: Signature

```
type Phase enumeration of ASLEEP, TIME, READY
automaton ClockSyncWLEst(n, f: Nat, ρ, β, δ, ϵ, P: Real,
                         i: Nat)
  signature
    input     start
    input     receive(m: Real ∪ {RDY}, j: Nat, const i: Nat)
                  where j ≠ i
    output    broadcast(m: Real ∪ {RDY}, const i: Nat)
    internal  update
```

- Phase is an enumerated type for the phase of the protocol
- ClockSyncWLEst is the specification for a single process in the protocol, each of which has a non-negative integer identifier
- Messages exchanged may be:
  - Either the time (a real)
  - Or the RDY message

TIOA For Clock Synchronization: States

```
states
  T: discrete AugmentedReal := ∞, // Local time for next round
  U: discrete Real := −1, // Local time for computing the adjustment
  V: discrete Real := −1, // Local time for broadcasting RDY message
  CORR: discrete Real := 0,
  Ph: Real,
  AV: Real := 0,
  DIFF: Array[Nat, discrete Real] := constant(0),
  phase: Phase := ASLEEP,
  RCVD_RDY: Set[Nat] := ∅,
  initially Ph ≥ 0
derived variables
  Local := Ph + CORR
```

`CORR` Correction value for computing local time from local physical clock

`Ph, Local` As in the clock maintenance algorithm

`DIFF` Estimated local time differences array, one element per process

`AV` Fault tolerant time average

`phase` Protocol's phase (type of last message sent)

`RCVD_RDY` Set with ids of processes from which a RDY was received

TIOA For Clock Synchronization: Initial States

The protocol does **NOT** assume that the local clocks are initially synchronized

- The **initially** clause just forces local clocks to be positive values
- The initial values for variables T, U, V ensure that the time triggered transitions will not be enabled or even scheduled in the initial states
- While phase = ASLEEP the automaton does not send any messages

TIOA For Clock Synchronization: Transitions (1/4)

```
input start
  eff
    T := Local
    phase := READY
```

- The start action changes the automaton from a passive state, in which it does not send messages, to an active state
- Immediately after, it will broadcast a time message

TIOA For Clock Synchronization: Transitions (2/4)

```
input receive(m,j,i)
  eff
    if ( m = RDY ) then
      RCVD_RDY := RCVD_RDY ∪ {j}
      if ( |RCVD_RDY| = n − f ) then
        for k: Nat where k < n do
          DIFF[k] := DIFF[k] − A
        od
        CORR := CORR + A
        T := Local
      endif
    else // Time message, with the local time at sender
      DIFF[j] = m + δ − Local
      if( phase = ASLEEP ) then
         T := Local
         phase := READY
      endif
    endif
```

TIOA For Clock Synchronization: Transitions (3/4)

```
output broadcast(m, i)
  pre
    ((Local = T ∧ phase = READY) ⇒ m = T)
    ∧ ( ((Local = V ∨ |RCVD–RDY| = (f + 1) ) ∧ phase = TIME )
        ⇒ m = RDY)
  eff
    if ( phase = READY ) then // TIME broadcast: start new round
      U := Local + (1+ρ)(2δ + 4ϵ)
      RCVD_RDY := ∅
      phase := TIME
    else // RDY broadcast
      phase := READY
    endif
```

- The phase = XXX in the premises of the implications on the preconditions ensure that a process does not send more than one message of the same type at a given time

TIOA For Clock Synchronization: Transitions (4/4)

internal update
  pre
    $Local = U \wedge Local > V$
  eff
    $AV := mid(reduce(DIFF))$
    $V := Local + (1 + \rho)(4\epsilon + 4\rho(8 + 2\epsilon) + 2\rho^2(\delta + 2\epsilon))$

- AV is now part of state, because it is computed at one time instant and applied at another
- AV is applied to the estimated local time differences rather than on the local time of the arrival of time messages
- The procedures mid() and reduce() are the same as in the clock synchronization maintenance algorithm

TIOA For Clock Synchronization: Trajectories

trajectories
  stop when
    $(\mathrm{Local} = \mathrm{T} \wedge \mathrm{phase} = \mathrm{READY}) \vee (\mathrm{Local} = \mathrm{U} \wedge \mathrm{Local} > \mathrm{V})$
    $\vee\ (\mathrm{Local} = \mathrm{V} \wedge \mathrm{phase} = \mathrm{TIME})$
  evolve
    $1/(1+\rho) \leq \mathbf{d}(\mathrm{Ph}) \leq 1 + \rho$

- Each of the terms in the **stop when** clause specifies a precondition of a transition that is time triggered
- Until the start transition, the physical clock is allowed to increase in an unconstrained way
- The occurrence of start forces the process to begin the first round
- In each round, the protocol will execute successively:
  1. the broadcast of a time message
  2. the update of its estimate of the correction value
  1. the broadcast of a RDY message

Synchronization Establishment Algorithm: Main Results

- Let B^i be the maximum difference between nonfaulty clock values at the latest real time when a nonfaulty process begins round i, i.e. when it broadcasts its clock value.
- As for the maintenance algorithm, the fault-tolerant averaging function used in the algorithm causes the difference to be approximately halved at each round

Lemma 20

For $i \geq 0, B^{i+1} \leq B^i/2 + 2\epsilon + 2\rho(11\delta + 39\epsilon)$

- Thus the limit of B^i as the round number increases without bound is $4\epsilon + 4\rho(11\delta + 39\epsilon)$, i.e. the algorithm achieves a closeness of synchronization of about 4ϵ.
- This protocol has two operation modes:
  1. To run it indefinitely
  2. To run it just until the desired closeness of synchronization is achieved and then to switch to the maintenance algorithm.
     - ★ In this case, we need for a protocol to switch between the two

Outline

Practice

- This algorithm has been adopted by FlexRay, the next-generation communications network protocol for the automotive domain.
- FlexRay uses a TDMA scheme in a broadcast medium.
  - Clock synchronization is used at a low-level to define the boundaries of the TDMA slots.
  - The protocol is supposed to support real-time applications, with rather small deadlines. I estimate that the synchronization of the clocks has to be in the range of μs.
- The authors report a problem with the protocols implementation on a broadcast medium:
  - If all nodes are very tightly synchronized, all of them may attempt to broadcast their time simultaneously causing collisions.
  - To avoid it, they propose staggering the broadcast times, so that process j broadcasts its i round times at its local clock time $T^i + j\sigma$, where σ is just big enough so that collisions are sufficiently infrequent that they can be attributed to faulty processors.
  - A worst-case analysis shows that the modified algorithm behaves very similarly to the original one.

Outline

Further Reading

- Jennifer Welch and Nancy Lynch, "A New Fault-Tolerant Algorithm for Clock Synchronization", Information and Computation 77, 1-36 (1988) (Available from Nancy Lynch's publications page on the Web.)
- Jennifer Lundelius, "Synchronizing Clocks in a Distributed System", S. M. thesis, MIT, MIT/LCS/TR-335 (Available on the Web. Use google to find it.)