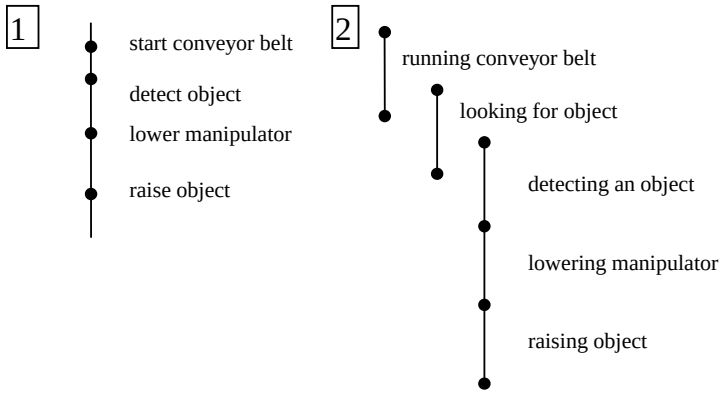
TIME HANDLING

Major issues:

- Representation
- Temporal Reasoning
- System time measurement and management

Time Representation

1. Point based - world view of the system has events that occur at a time instant, results in a state change, and takes zero time to occur.
2. Time-interval based - activities take a finite amount of time and have associated start and stop times.



Time point representation has the disadvantage that instantaneous events are not decomposable into sub-events while retaining the event ordering. The timer interval approach allows partially overlapping relations but may also admit a cumulative loss of time if not properly managed, e.g. clock interrupt to actual store delay.

A Real-Time system needs both representation types - time point based for deadline specification and interval based for process computational time specification.

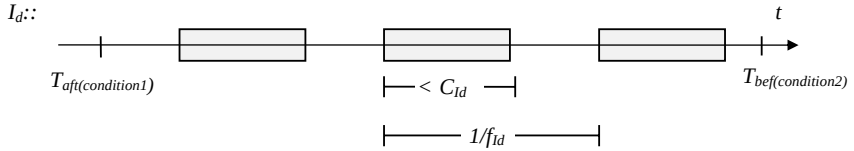
Time Constraints

Time constraints can be viewed as requirements on processes to start executing after satisfying their start conditions, and to complete execution before the respective deadlines. An extension is to include periodic executions of processes with finite execution time intervals.

Formally, we can describe the time constraint as a 5-tuple:

$(I_d, T_{aft(condition1)}, C_{Id}, f_{Id}, T_{bef(condition2)})$

where: I_d is the executable process or object name
$T_{aft(condition1)}$ is the event after which execution of I_d begins
C_{Id} is the bound on the computation time of each instance of I_d
f_{Id} is the frequency with which the computation is to occur
$T_{bef(condition2)}$ is the deadline before which execution of I_d must terminate



- The time interval $T_{bef(condition2)} - T_{aft(condition1)}$ is known as the *occurrence interval*.
- Where no deadline constraint is imposed we have $C_{Id} = \infty$ (or an *off-line* computation situation in contrast to *on-line* computation).
- In this simple time constraint model, the bounds are assumed to be deterministic and are derived from requirements external to the object.

Time service and synchronization

In a distributed system each site or node in the network should have access to a source of time knowledge - i.e. a *clock*. It is useful to examine the way in which time knowledge can be characterised and communicated.

Definitions:

1. A *standard* or *reference clock i* has $\forall t$: $C_i(t) = t$

2. A clock *i* is *correct* at time t_o if $C_i(t_o) = t_o$

3. A clock *i* is *accurate* at time t_o if $\left.\frac{dC_i(t)}{dt}\right|_{t=t_o} = 1$

A clock *drifts* if it is inaccurate at some time.

Clock synchronization: used for event ordering purposes and enhancement of time knowledge. A clock update during synchronization can be expressed as:

$$C_i(t_i) \leftarrow F\,[C_{i1}(t_{i1}), C_{i2}(t_{i2}), \ldots, C_{ik}(t_{ik})]$$

where $C_{i1}(t_{i1}), C_{i1}(t_{i1}), \ldots, C_{ik}(t_{ik})$ are *k* clocks that are used to synchronize $C_i(\bullet)$ through some algorithm *F* (which ideally should be *monotonic* to preserve local event ordering).

Real-time systems require structured time ordering mechanisms to reason about events - an *event* is a detectable, instantaneous and atomic change in *system state*. The *system state* includes the set of clocks $\{C_i(\bullet)\}$

Define $\{C_i(t) = C_o\}$ as the set of system states for which the clock *i* has time C_o then the following predicates can be defined:

$T_{aft}(C_o) = true$ if $C_i(t) \geq C_o$, otherwise *false*
$T_{bef}(C_d) = true$ if $C_i(t) \leq C_d$, otherwise *false*

Types of Clock Systems

1. *Central*: one accurate clock, possibly with a standby for fault tolerance; special purpose hardware to handle requests from any executing process.

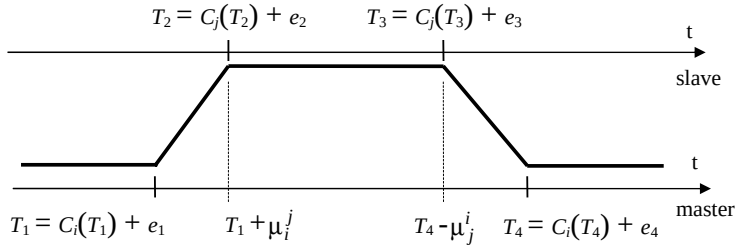
2. *Centrally controlled*: master clock polls slave clocks and clock differences are used to correct slaves; master clock failure $\rightarrow$ new master clock.

3. *Distributed*: all sites homogeneous, updating their own clocks after receipt of time from all other clocks; fault tolerance is protocol based $\rightarrow$ no side effects of site failure but communications traffic is heavier.

Centrally controlled clock system algorithms

Also called *master-slave* clock systems. Suppose we have a master clock *i* starting a clock synchronization procedure at time T_1 - it has current clock value $C_i(T_1)$ with error e_1.

The clock value is sent to slave *j*, takes μ_i^j time units to reach *j* and is received at time T_2. At that time the slave clock has a value $C_j(T_2)$ with error e_2:



At time T_2 we have: $C_i(T_1) + e_1 + \mu_i^j = C_j(T_2) + e_2 = T_2$

and the slave can compute a difference: $d_1 = C_j(T_2) - C_i(T_1)$

$$= \mu_i^j + e_1 - e_2$$

Suppose that the error difference (i.e. $e_1 - e_2$) can be modelled as a clock skew of the slave ξ_j with a zero-mean random noise process E_j^1, i.e:

$$d_1 = \mu_i^j + \xi_j - E_j^1$$

The same process is repeated in the reverse direction, i.e. the slave reads a clock value of $C_j(T_3)$ at time T_3 and sends it to the master with the calculated difference d_1. The message has travel time μ_j^i and is received at T_4 where the master reads $C_i(T_4)$ with error e_4.

The master now computes $d_2 = C_i(T_4) - C_j(T_3)$

$= \mu^i_j + e_3 - e_4 = \mu^i_j - \xi_j - E^2_j$

where $e_4 - e_3$ is the same slave clock skew ξ_j and E^2_j is another noise process.

From these values of d_1 and d_2 we have:

$(d_1 - d_2)/2 = [(\mu^j_i + \xi_j - E^1_j) - (\mu^i_j - E^2_j - \xi_j)]/2$

$= \xi_j + (\mu^j_i - \mu^i_j)/2 - (E^1_j - E^2_j)/2$

A number of successive polls can be taken to produce:

$\bar{\xi}_j = \xi_j + (\bar{\mu}^j_i - \bar{\mu}^i_j)/2 - (\bar{E}^1_j - \bar{E}^2_j)/2$

where $\mu^j_i \approx \mu^i_j$ and the noise processes E^k_j are symmetrically distributed, then $\xi_j \cong \bar{\xi}_j$ which is the average clock skew which can be used to correct the clock $C_j(\bullet)$. Better knowledge about μ^j_i and μ^i_j can also be used to improve the estimate.

Suppose the clock is synchronized with this algorithm at least every τ seconds and the error is given by:

$\epsilon_j = (\bar{\mu}^j_i - \bar{\mu}^i_j)/2 - (\bar{E}^1_j - \bar{E}^2_j)/2$

Also suppose the clock for each slave has a maximal drift rate of δ_j. A bound on the *maximal* clock difference for any slave is given by:

$$\left|2\tau \max_j(\delta_j)\right| + \left|2 \max_j(\epsilon_j)\right|$$

i.e. the update frequency ($1/\tau$) controls the bounds on synchronization correctness between updates.

The imposed communication load for n processors and p polls is $(2p+1)n$ messages.

Compensation for the communication delays can be partially accomplished by separating the delays into predictable and unpredictable components: $\mu^j_i = \bar{\mu}^j_i + \Delta\mu^j_i$.

This algorithm is the basis for the TEMPO algorithm used in 4.3BSD UNIX and the ICMP protocol. Predictable communication delay compensation can be obtained from the ICMP routing timestamp option field.

Distributed Clock Algorithms

This approach offers increased fault tolerance at the expense of an increased communication load. There are numerous examples of this type of algorithm, but to illustrate we only need to look at the two major components:

- fundamental ordering
- accuracy enhancement

Fundamental ordering: this approach is based on message timestamping with the following properties:

- the accuracy of clock i is bounded by a drift rate δ:
$$\left|1 - \frac{dC_i(t)}{dt}\right| < \delta << 1$$
- the communication interconnection graph of the computation nodes is closely connected with diameter d.
- the network imposes an unpredictable, but bounded, message delay D, i.e. $\mu < D < \eta$ where μ and η are the bounds.

Algorithm:

- the local clock is incremented on each local time event:
$C_i(t) \leftarrow C_i(t)+1$
- each process with a clock sends a message to the others every τ seconds (at least) and initiates a timestamp T_m.
- on receipt of a external T_m the receiver sets its clock to:
$C_i(t) \leftarrow \max(C_i(t), T_m + \mu)$

The communication cost of an update is $n(n\text{-}1)$ messages for n clocks. A bound on the variation of each clock can be written as:

$$\forall i \forall j: \quad \left| C_i(t) - C_j(t) \right| < d(2\delta\tau + \eta)$$

Note that this algorithm only achieves an ordering goal $\rightarrow$ clock differences are bounded without improvement.

Time intervals - minimize maximum error: this approach uses knowledge of the bounds on the error of the clocks with the following assumptions:

- every clock i is known to be correct within the interval:

  $[C_i(t) - E_i(t), C_i(t) + E_i(t)]$

  where $E_i(t)$ is a bound on the error of' clock i.

- the error interval is constructed from the following:
  - the error (made up of discretization and other constant errors) denoted by $\in_i$ has effect at the clock *reset* time denoted by ρ_i.
  - the *delay* from clock i being read until another clock j uses the readout for its update μ_i^j.
  - the degradation (or drift) in time that develops between consecutive resets is δ_i.

The algorithm itself has two rules: a *response rule* and a *synchronizer rule*, i.e. a request transmitted by the synchronizer rule at node j activates a response from node i:

The response rule from node $i \neq j$ is:

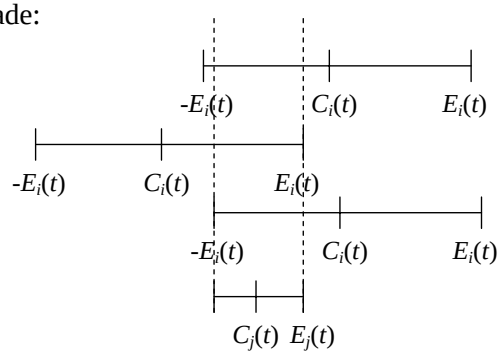
- $E_i(t) \leftarrow \in_i + [C_i(t) - \rho_i]\delta_i$
- Send $[C_i(t), E_i(t)]$ to node j
- where the modified error bound $E_i(t)$ indicates an interval in which node i's clock is correct.

The synchronizer rule is repeated at least every τ time units and is as follows:

- $\forall i \neq j$: request $[C_i(t), E_i(t)]$ from node i.
- for each $[C_i(t), E_i(t)]$ received, check that there is a non-empty intersection in the intervals $[C_i(t) - E_i(t), C_i(t) + E_i(t)]$ and $[C_j(t) - E_j(t), C_j(t) + E_j(t)]$, otherwise ignore this node.
- if $E_i(t) + (1+\delta_j)\mu_i^j \leq E_j(t)$, i.e. the error of the response plus the response delay produce an error smaller than the local error, otherwise ignore this node.
- if all the above conditions hold, then the synchronizer at node j can reset its clock and improve its time knowledge via:

  $C_j(t) \leftarrow C_i(t)$ -- clock update
  $\in_j \leftarrow E_i(t) + (1+\delta_j)\mu_i^j$ -- error update
  $\rho_j \leftarrow C_i(t)$ -- reset time update

A related *intersection* algorithm uses a modified synchronizer rule, i.e. the intersection of all the response time intervals $[C_i(t) - E_i(t), C_i(t) + E_i(t)]$ is taken - clock update is then to the interval's midpoint, and the error updated to half the interval. If there is no intersection interval then no update is made:



The intersection algorithm has superior accuracy to the minimize maximum error algorithm, but has poorer fault tolerance.