Distributed Systems (CS543)

Time and Global States

Dongman Lee

KAIST

Class Overview

- Synchronization in Distributed Systems
- Physical and Logical Time
- Global States

Time in Distributed Systems

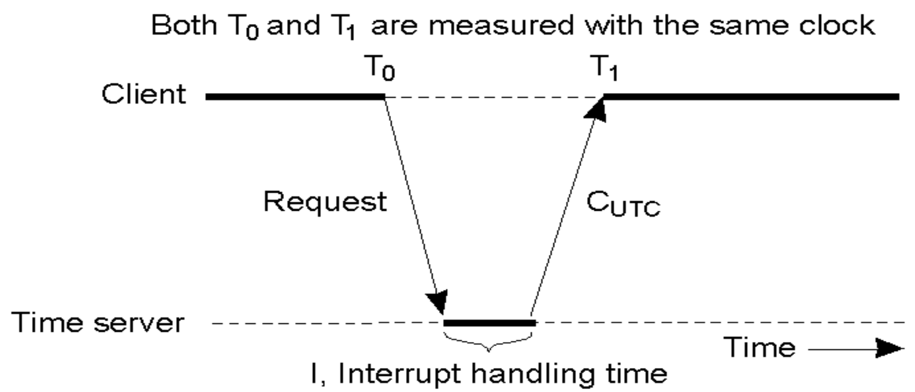
- In a distributed system
  - time is used in many applications
    - data consistency
    - security
    - causality of events
  - to maintain a global state
    - states are shared only via message passing
    - need to know in which order events happened
  - each system has its own clock
    - no global clock
    - local clocks may be out of sync
  - ➲ ordering based on time
    - physical time
    - logical time

Physical Clock Synchronization

- Clock synchronization
  - external synchronization
    - UTC (Coordinated universal time)
  - internal synchronization
    - Cristian’s algorithm
    - Berkeley algorithm
    - NTP

Cristian's Algorithm

- Assumptions
  - There exists a central time server, receiving signals from a source of UTC
  - Process P sets its clock by communicating with the time server
- Algorithm
  - P sends a time request message to the server and gets a reply with time t from it
  - P sets its clock to $t + T_{trans}$ where T_{trans} is time taken to transmit reply from the server to P
    - use $\pm(T_{round}/2 - min)$ as accuracy of T_{reply} if $T_{trans} = min + x$ $(x >= 0)$

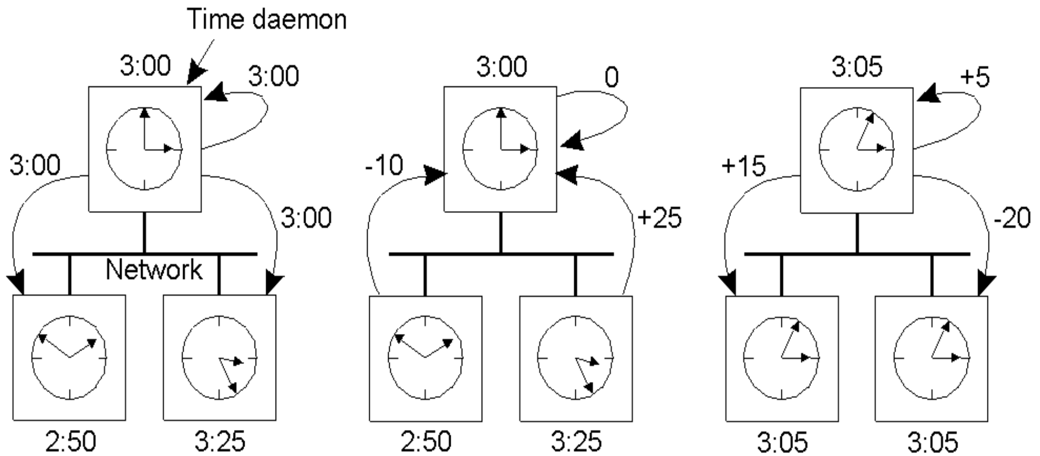


Cristian’s Algorithm (cont.)

- Problems
  - a single point of failure
  - dishonest time server

Berkeley Algorithm

- Assumptions
  - No UTC receiver exists
  - one computer is chosen to act as *master*, others *slaves*
- Algorithm
  - master periodically collects each slave's time by observing round trip time and averages times (fault-tolerant average)
    - only subset of clocks is chosen which do not differ from more than a specified time
  - master sends adjustment to each slave
  - new master is elected in presence of failure of current master

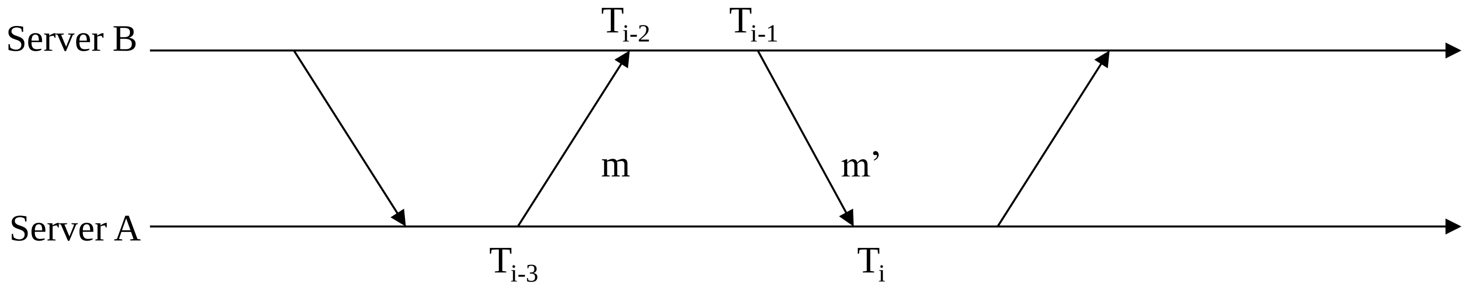


Network Time Protocol

- Assumptions
  - servers are connected in a logical hierarchy called synchronization subnet
  - primary (top level) servers are directly connected to time source (e.g. UTC)
  - NTP servers synchronized by one of three modes
    - multicast
      - one or more servers periodically multicast the time to workstations assuming the delay in a few milliseconds
      - relatively low accuracy
    - procedure call
      - similar to Cristian's algorithm
      - one time server plays a role of client and the other a role of server
    - symmetric
      - a pair of servers exchange messages to synchronize their clocks, forming an association

Network Time Protocol

- Algorithm
  - For each pair of messages sent between two servers the NTP protocol calculates
    - offset o_i: estimate of actual offset between two clocks
    - delay d_i: total transmission time for two messages



o: true offset of clock at B relative to that at A
t: actual transmission time for m
t': actual transmission time for m'

$t = a - o$ where $a = T_{i-2} - T_{i-3}$
$t' = -b + o$ where $b = T_{i-1} - T_i$
$d_i = t + t' = a - b$ and $o_i = (a + b)/2$
$\Rightarrow b = (a+b)/2 - (a-b)/2 <= o <= (a+b)/2 + (a-b)/2 = a$
$\Rightarrow o_i - d_i/2 <= o <= o_i + d_i/2$

Physical Clock Synchronization Problem

- Synchronization is necessarily inaccurate
- can get out of sync if network is partitioned
- vulnerable to malicious time servers

Logical Time

- Happened-before [Lamport] relation
  - if two events occurred at the same process and A executed before B, then A -> B
  - if event A is sending message and B is receiving message, then A -> B
  - A -> B, B -> C, then A -> C
  - if A and B are not ordered, they are concurrent, A || B
- Notes
  - happened-before relation only captures potential causality
    - no guarantee of real connection between A and B even if A -> B

Logical Timestamp

- Goal: how to capture “happened-before relation” while allowing concurrent events
- Solutions
  - LC1:
    - Cp is incremented before each event is issued at P: Cp = Cp + 1
  - LC2:
    - when P sends a message *m*, *t* = Cp on *m*
    - on receiving (*m*, *t*), Q computes Cq = max (Cq, *t*) and sets its clock to Cq+1
  - if A -> B, then C(A) < C(B) but not vice versa

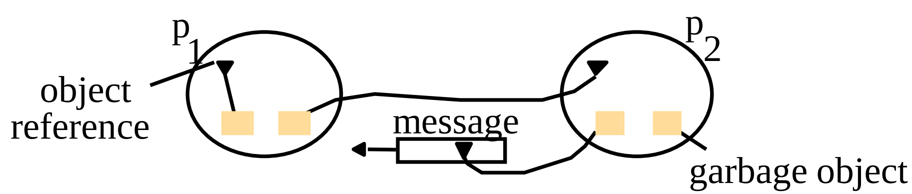
Total Event Ordering

- Logical clock imposes only a partial order
  - order all the events at which they occur => total event ordering
- Solutions
  - represent a logical time as a pair of timestamp and process id
    - (T_a, P_a)
  - arbitrary total ordering of processes is used
  - if a is an event at process P_i and b is an event at process P_j, then $a \Rightarrow b$ iff either (i) $C_i(a) < C_j(b)$ or (ii) $C_i(a) = C_j(b)$ and $P_i < P_j$

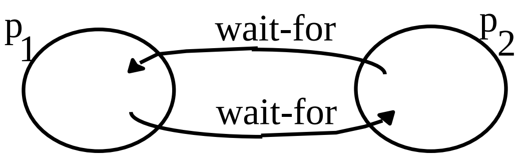
Global States

- Why global states?
  - due to separation – communication delay and relative speeds of computations (i.e. little possibility of achieving perfect clock synchronization), it is difficult to tell whether a system works consistently simply by looking at the collection of the individual process histories in which one process executes some notification or reaction when the state of the system satisfies a particular condition

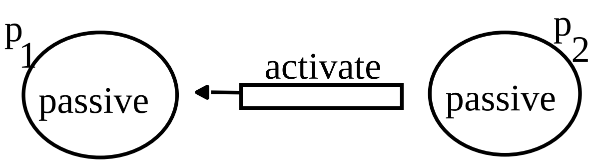
a. Garbage collection



b. Deadlock

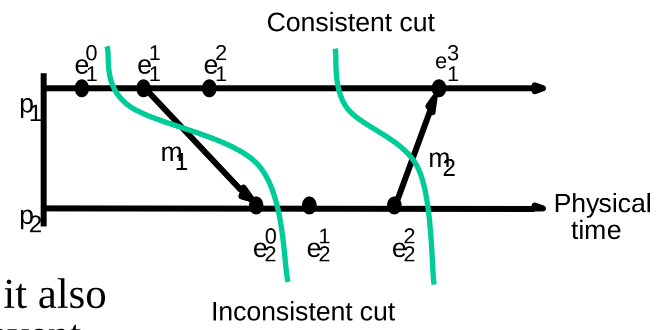


c. Termination



Global States (cont.)

- Definitions
  - *global history*
    - the union of the individual process histories
  - *cut*
    - a subset of the system's global history
    - a cut C is consistent if, for each event it contains, it also contains all the events that happened-before that event
  - *consistent global state*
    - a global state that corresponds to a consistent cut
    - the execution of a distributed system can be considered a series of transitions between global states of the system
  - *run*
    - a total ordering of all the events in a global history that is consistent with each local history's ordering
  - *linearization* or *consistent run*
    - an ordering of the events in a global history that is consistent with happened-before relationship
    - S' is *reachable* if there is linearization that passes through S and then S'



Global States (cont.)

- Global state predicate
  - a function that maps from the set of global states of processes in the system to {true, false}
  - *stable*
    - once the system enters a state in which the predicate is true, it remains true in all future states reachable from that state
  - two properties
    - safety
      - the system always remains in a certain state – partial correctness specifications
    - liveness
      - something eventually happens – total correctness specification
      - e.g. termination

Global States (cont.)

- Chandy and Lamport's snapshot algorithm
  - algorithm to determine global states of distributed systems
    - states are recorded locally and collected by a designated server
  - assumptions
    - no failures in channels and processes – exactly once delivery
    - unidirectional channel and FIFO ordered message delivery
    - always a path between any two processes
    - global snapshot initiation at any process at any time
    - no process activity halt during snapshot

Global States (cont.)

- Chandy and Lamport's snapshot algorithm (cont.)

Marker receiving rule for process p_i

On p_i's receipt of a *marker* message over channel *c*:

if (p_i has not yet recorded its state) it

records its process state now;

records the state of *c* as the empty set;

turns on recording of messages arriving over other incoming channels;

else

p_i records the state of *c* as the set of messages it has received over *c* since it saved its state.

end if

Marker sending rule for process p_i

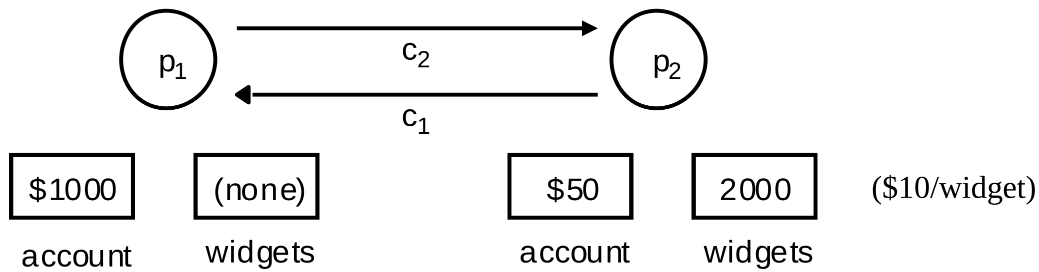
After p_i has recorded its state, for each outgoing channel *c*:

p_i sends one marker message over *c*

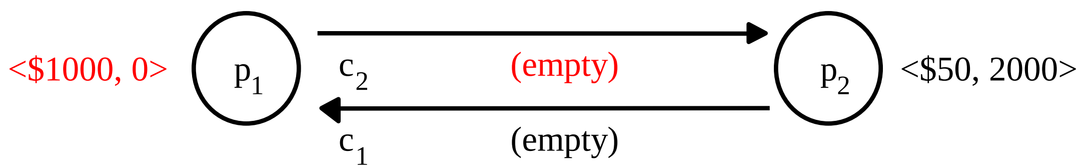
(before it sends any other message over *c*).

Global States (cont.)

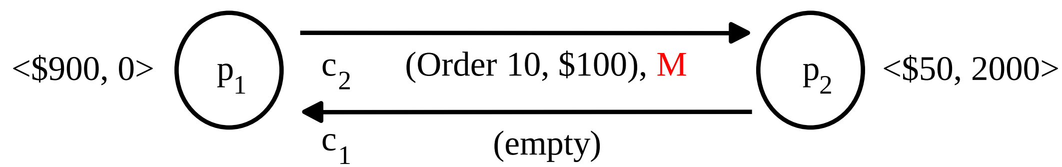
- Example



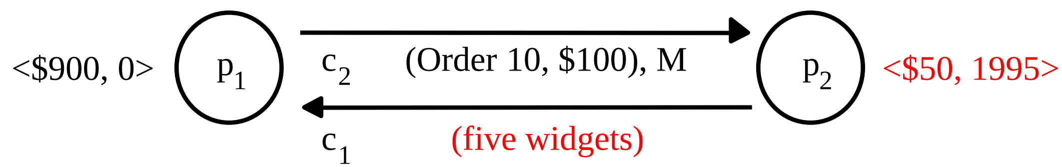
1. Global state S_0



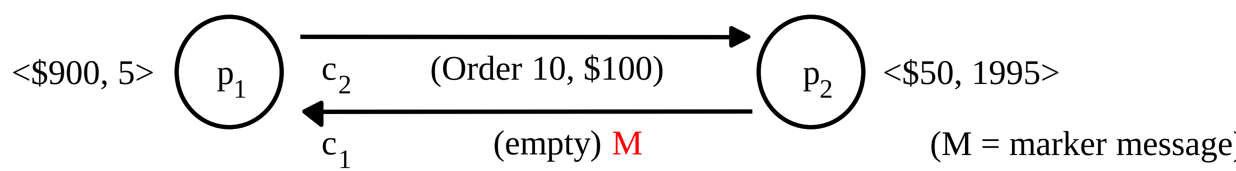
2. Global state S_1



3. Global state S_2



4. Global state S_3



Global States (cont.)

- Stability and reachability between states in snapshot algorithm
  - if a stable predicate is true in the state S_{snap} then we may conclude that the predicate is true in the state S_{final} by using reachability