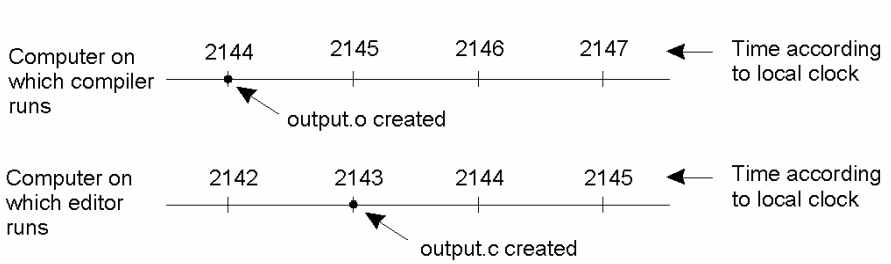
Synchronization

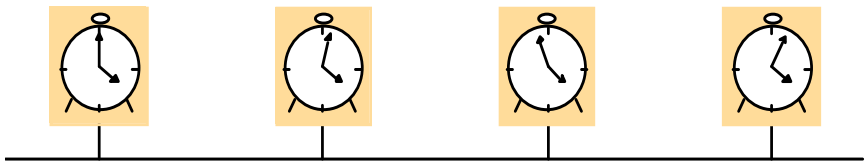
Distributed Software Systems

Clock Synchronization

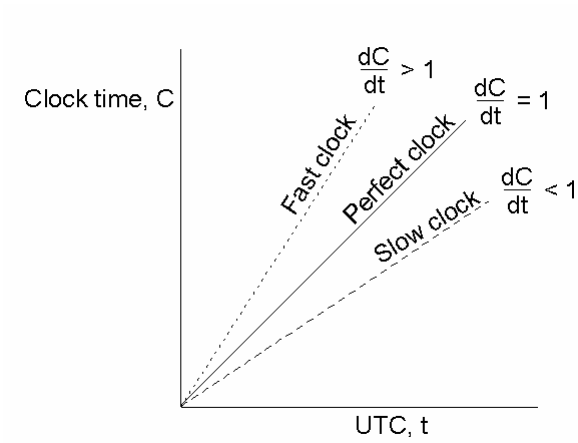


When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Skew between computer clocks in a distributed system



Clock Synchronization Algorithms



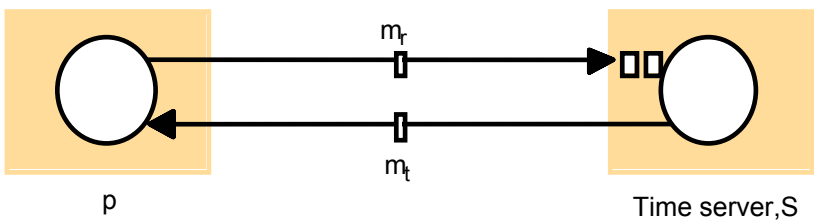
The relation between clock time and UTC when clocks tick at different rates.

Clock Synchronization

- Physical clocks drift, therefore need for clock synchronization algorithms
  - Many algorithms depend upon clock synchronization
  - Clock synch. Algorithms – Christian, NTP, Berkeley algorithm, etc.
- However, since we cannot perfectly synchronize clocks across computers, we cannot use physical time to order events

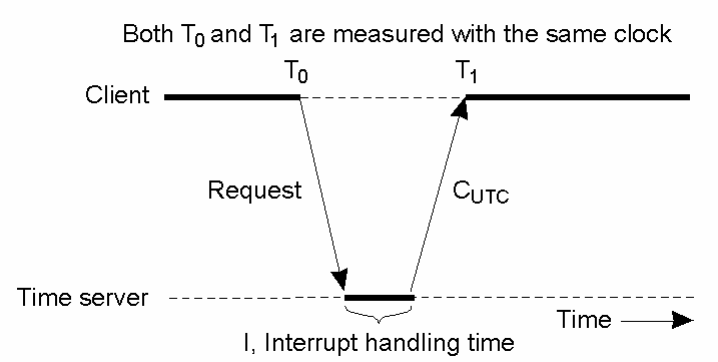
5

Clock synchronization using a time server



6

Cristian's Algorithm

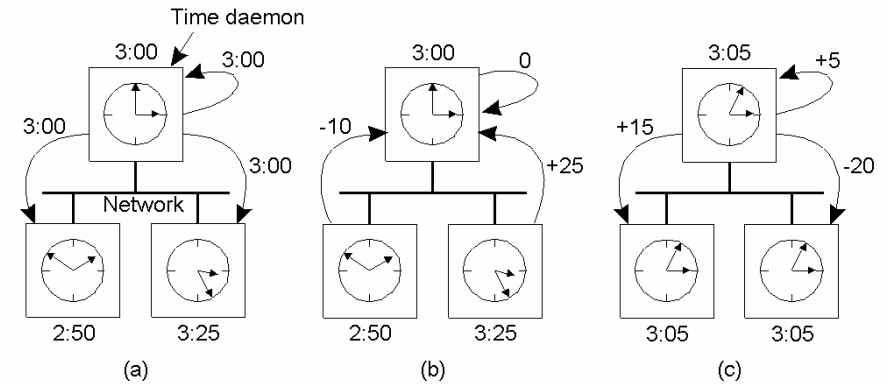


Getting the current time from a time server.

Clock synchronization algorithms

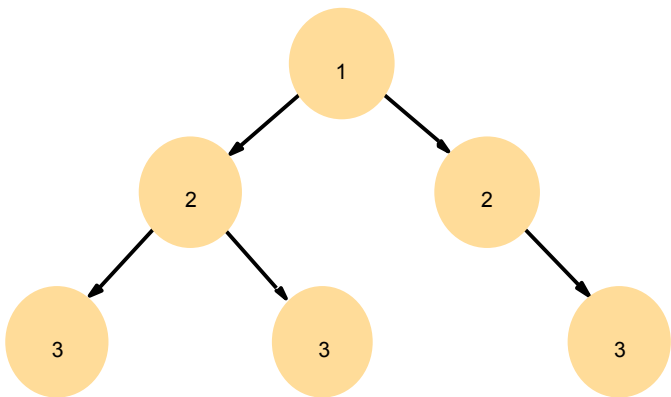
- Cristian's algorithm
  - p should set its time to t + $T_{round}/2$
  - Earliest time at which S could have placed its time in m_t was min after p dispatched m_r
  - Latest point at which it could do so was min before m_t arrived at p
  - Time by S's clock when message arrives at p is in range [t + min, t + T_{round} – min]
    - Accuracy $\pm(T_{round}/2\text{-min})$

The Berkeley Algorithm



a) The time daemon asks all the other machines for their clock values
b) The machines answer
c) The time daemon tells everyone how to adjust their clock

An example synchronization subnet in an NTP implementation

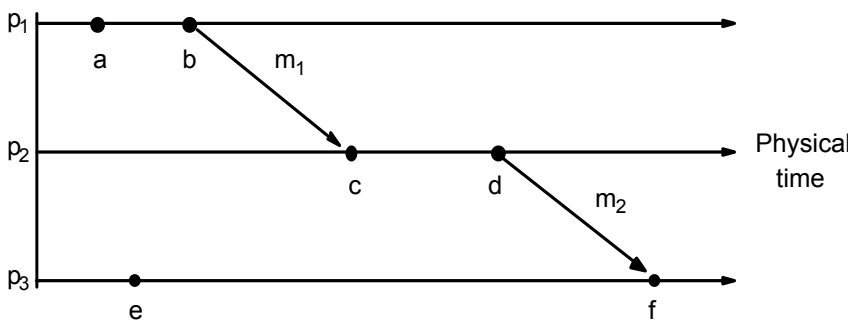


Note: Arrows denote synchronization control, numbers denote strata.

Logical time & clocks

- Lamport proposed using logical clocks based upon the "happened before" relation
  - If two events occur at the same process, then they occurred in the order observed
  - Whenever a message is sent between processes, the event of sending occurred before the event of receiving
  - X happened before Y denoted by $X \rightarrow Y$

Events occurring at three processes

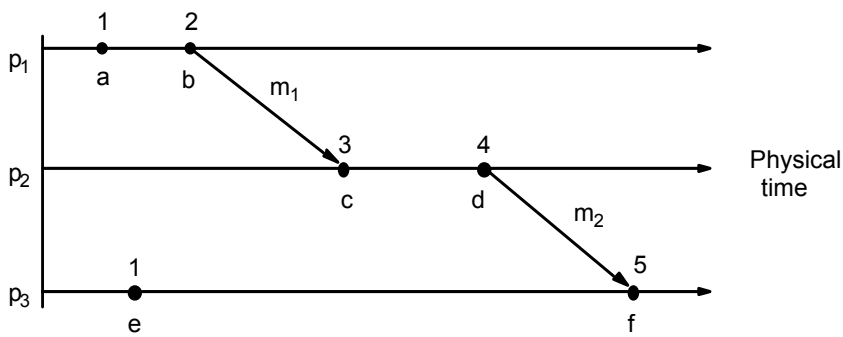


Lamport's algorithm

- Each process has its own logical clock
- LC1: C_p is incremented before each event at process p
- LC2:
  1. When process p sends a message it piggybacks on it the value C_p
  2. On receiving a message (m,t) a process q computes $C_q = \max(C_q, t)$ and then applies LC1 before timestamping the receive event

13

Lamport timestamps for the events



14

Vector Timestamps

- Shortcoming of Lamport's clocks: if L(e) < L(f), we cannot conclude that e → f
- Vector clocks
  - A process keeps an array of clocks, one for each process
  - Like Lamport timestamps, processes piggyback vector timestamps on messages they send each other

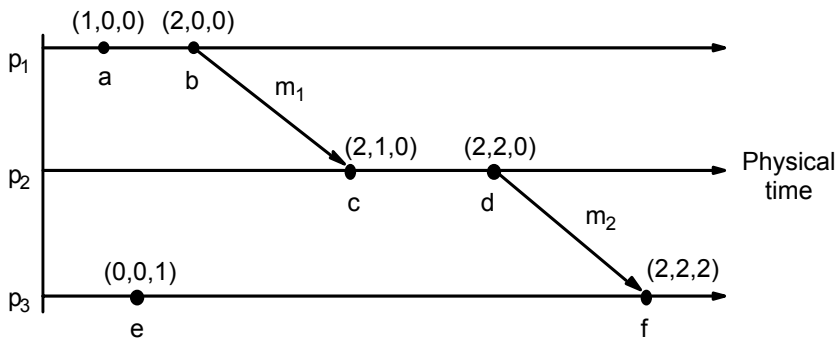
    V = W iff V[j] = W[j] for j = 1, 2, …, N

    V ≤ W iff V[j] ≤ W[j] for j = 1, 2, …, N

    V < W if V ≤ W and V ≠ W

15

Vector timestamps for the events



16

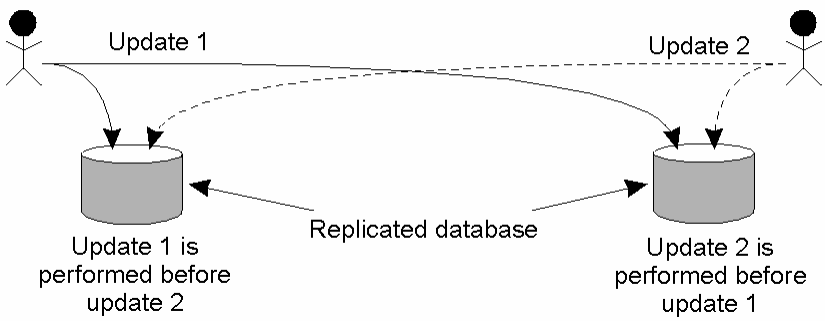
Totally ordered logical clocks

- Logical clocks only impose partial ordering
- For total order, use (T_a,P_a) where P_a is processor id
- $(T_a,P_a) < (T_b,P_b)$ if and only if either $T_a < T_b$ or $(T_a = T_b$ and $P_a < P_b)$
- This ordering has no physical significance, but it is sometime useful, e.g. to break a tie between two processes trying to enter a critical section

17

Example: Totally-Ordered Multicasting

Updating a replicated database and leaving it in an inconsistent state.



18

Distributed mutual exclusion

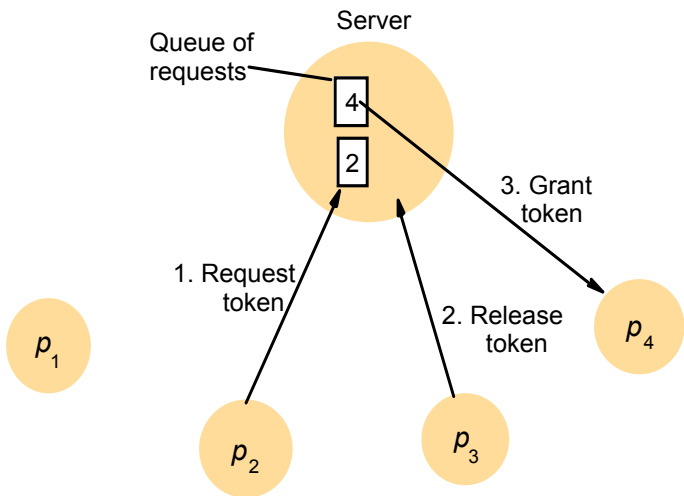
- Central server algorithm
- Ricart and Agrawal algorithm
  - A distributed algorithm that uses logical clocks
- Ring-based algorithms

NOTE: the above algorithms are not fault-tolerant and not very practical. However, they illustrate issues in the design of distributed algorithms

- Several other mutual exclusion algorithms have been proposed
  - Quorum consensus algorithms – Maekawa's algorithm
  - We will discuss majority voting in the context of replicated data management

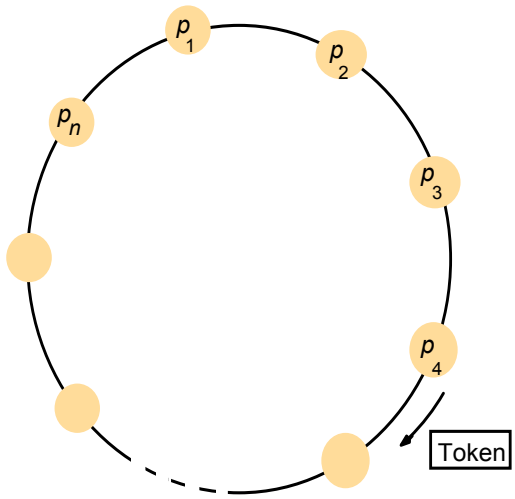
19

Server managing a mutual exclusion token for a set of processes



20

A ring of processes transferring a mutual exclusion token



21

Ricart and Agrawala's algorithm

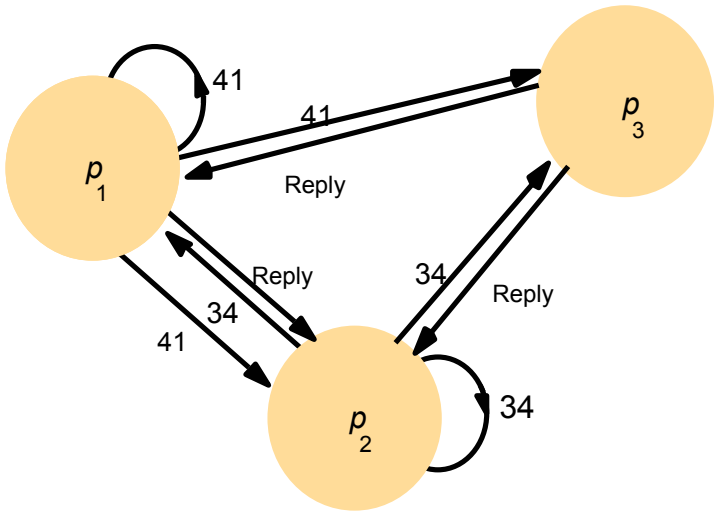
On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast *request* to all processes;
    T := request's timestamp;
    Wait until (number of replies received = (*N* – 1));
    state := HELD;

request processing deferred here

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
    if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;

22

Multicast synchronization



Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

A comparison of three mutual exclusion algorithms.

Maekawa's algorithm

- Every node needs permission from other nodes in its quorum before it enters critical section
- Quorums are constructed in such a way that no two nodes can be in their critical section at the same time
- The size of each node's quorum is $O(\sqrt{N})$, which can be shown to be optimal

25

Construction of quorum sets

Consider a system with 9 nodes

The quorum for any node includes the nodes in its row and column

1	2	3
4	5	6
7	8	9

Quorum for Node 1 = {1,2,3,4,7}

Quorum for Node 2 = {2,5,8,1,3}

There is a non-null intersection for the quorums of any two nodes

26

Maekawa's algorithm

On initialization
    state := RELEASED;
    voted := FALSE;
For p_i *to enter the critical section*
    state := WANTED;
    Multicast *request* to all processes in $V_i - \{p_i\}$;
    Wait until (number of replies received = $(K - 1)$);
    state := HELD;
On receipt of a request from p_i *at* p_j $(i \neq j)$
    if (*state* = HELD *or voted* = TRUE)
    then
        queue *request* from p_i without replying;
    else
        send *reply* to p_i;
        voted := TRUE;
    end if

Maekawa's algorithm – cont'd

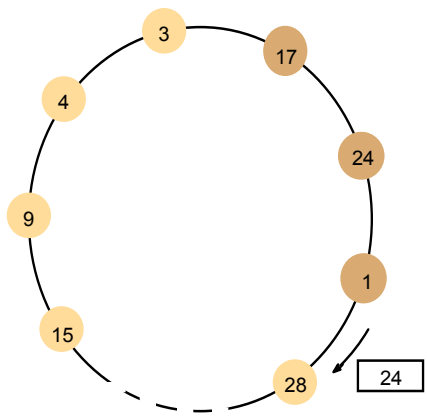
For p_i *to exit the critical section*
    state := RELEASED;
    Multicast *release* to all processes in $V_i - \{p_i\}$;
On receipt of a release from p_i *at* p_j $(i \neq j)$
    if (queue of requests is non-empty)
    then
        remove head of queue – from p_k, say;
        send *reply* to p_k;
        voted := TRUE;
    else
        voted := FALSE;
    end if

Election Algorithms

- An election is a procedure carried out to chose a process from a group, for example to take over the role of a process that has failed
- Main requirement: elected process should be unique even if several processes start an election simultaneously
- Algorithms:
  - Bully algorithm: assumes all processes know the identities and addresses of all the other processes
  - Ring-based election: processes need to know only addresses of their immediate neighbors

29

A ring-based election in progress

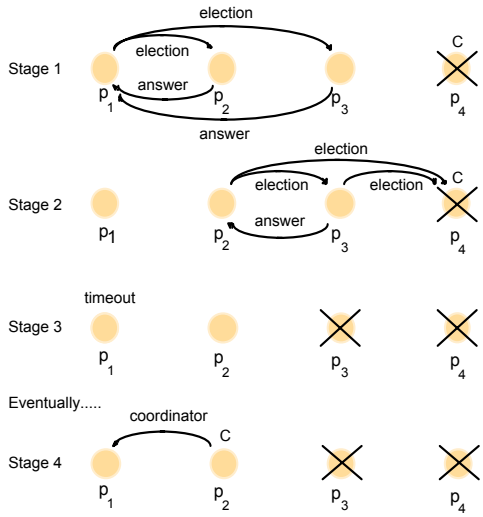


Note: The election was started by process 17.
The highest process identifier encountered so far is 24.
Participant processes are shown darkened

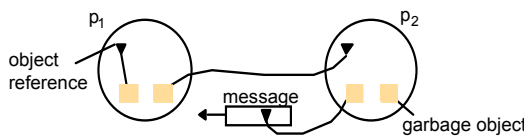
30

The bully algorithm

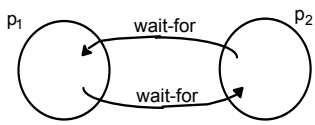
The election of coordinator p_2, after the failure of p_4 and then p_3



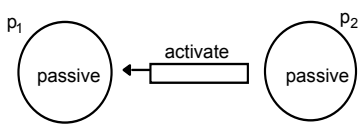
Detecting global properties



a. Garbage collection



b. Deadlock



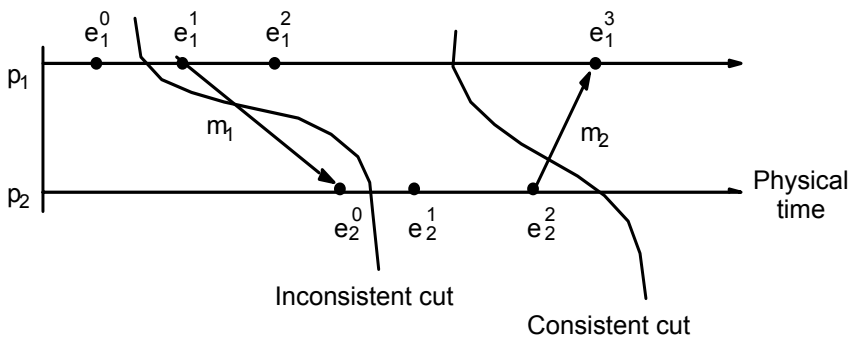
c. Termination

Global states and consistent cuts

- Capturing a global state would be straightforward if we had perfectly synchronized clocks
- How to capture a meaningful global state from local states recorded at different real times?
- Each process records events that correspond to internal actions, e.g. updating a variable, and the sending or receipt of a message
- A Cut is a subset of the system's global history that is a union of prefixes of process histories
- A cut is **consistent** if for each event it contains, it also contains all events that happened before that event

33

Consistent and Inconsistent Cuts

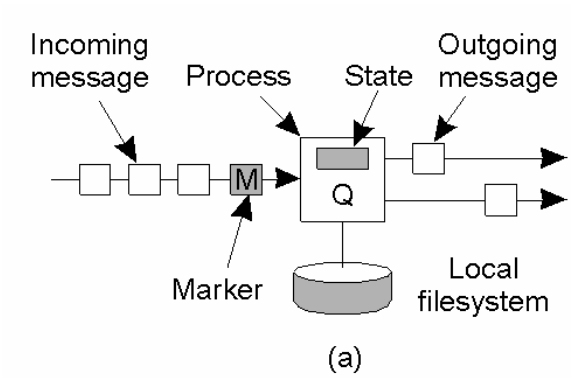


34

Chandy and Lamport's snapshot algorithm

- Goal: record a set of process and channel states for a set of processes such that even if the combination of recorded states may never have occurred at the same time, the recorded state is consistent
- State recorded locally at processes
- Assumptions
  - neither channels nor processes fail; communication is reliable
  - channels are unidirectional and provide FIFO message delivery
  - the graph of processes and channels is strongly connected
  - any process may initiate a global snapshot at any time
  - processes may continue with their execution and send and receive normal messages while the snapshot takes place

Chandy and Lamport algorithm



Organization of a process and channels for a distributed snapshot

Chandy and Lamport's 'snapshot' algorithm

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

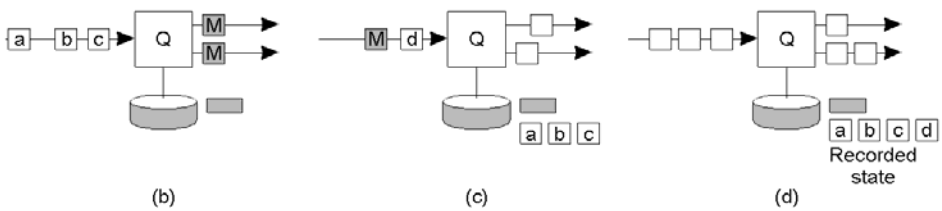
Marker sending rule for process p_i

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).

Chandy & Lamport algorithm

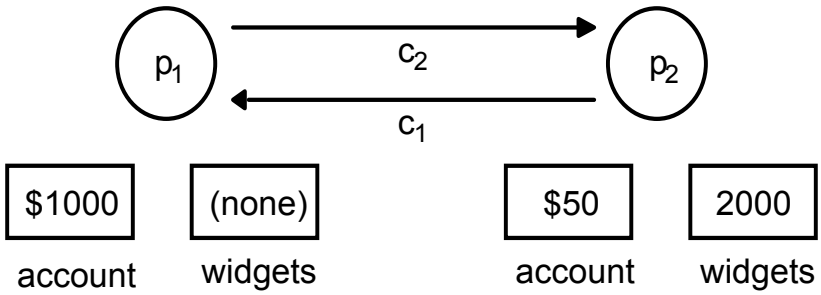


b) Process Q receives a marker for the first time and records its local state

c) Q records all incoming message

d) Q receives a marker for its incoming channel and finishes recording the state of the incoming channel

Two processes and their initial states

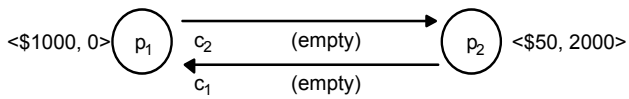


Assume that p_2 has already received an order for five widgets, which it will shortly dispatch to p_1

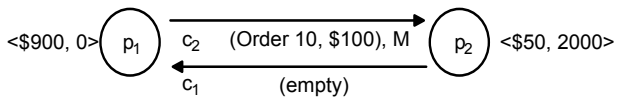
39

Example: Chandy & Lamport's algorithm

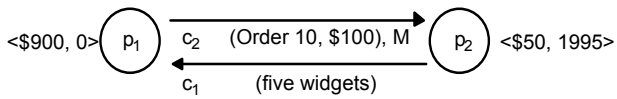
1. Global state S_0



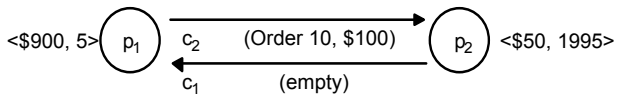
2. Global state S_1



3. Global state S_2



4. Global state S_3



(M = marker message)

40