CS G623: Advanced Operating Systems

Lecture 4-6

BITS Pilani
Pilani Campus

Amit Dua

August 9, 2018

Topics already discussed

- Advanced OS
- Types of AOS
- Motivation for Distributed systems
- Components of distributed system
- Goals
  - Access remote resources
    - Resource
    - Why collaboration
    - Price for collaboration
  - Transparency
    - Acc, loca, reloca, migra, repli, concurr, security
    - limitations
  - Open
    - Interface specification complete, neutral
    - Interoperability, portability, extensible
  - Scalable
    - Size, geography, administration
    - Central server, LAN, different organizations
    - Hiding latency, distribution, replication
    - Asynchronous, client side, DNS, Internet, caching, cost of concurrent updation
- Types of DS
  - Cluster and grid computing systems

Topics already discussed

- Issues with DS
- Message passing vs Remote procedure calls (RPC)

Topics to be discussed

- Global clock
- Logical clock
- Vector clock
- Causal ordering of messages
- BSS
- SES
- Global state recording
- Cuts
- Termination detection

Distributed systems limitation

- Absence of a global clock
  - Possible solutions
    1. Common clock for all distributed computers
       - Disadvantage: Unpredictable and variable transmission delays make it impractical
    2. Synchronized clocks, one for each computer
       - Disadvantage: Each clock will drift at a different rate, making it impractical
  - Conclusion
    - No system-wide physical common (global) clock can be implemented
  - Consequences
    - Temporal ordering of events is difficult (e.g., scheduling)
    - Collecting up to date information is difficult
- Absence of shared memory
  - No single process can have complete, up-to-date state of entire distributed system (global state)

Distributed systems limitations (cont.)

- Any operating system or process cannot know accurately the current state of all processes in the distributed system
- An operating system or process can only know
  - The current state of all processes on the local system
  - The state of remote operating systems and processes that is received by messages
    - These messages represent the state in the past

Terminology

Channel
- Exists between two processes if they exchange messages
- Each channel is unidirectional

State
- Sequence of messages that have been sent and received along channels incident with the process

Snapshot
- Records the state of a process
- Includes a record of all messages sent and received on all channels since the last snapshot

Global state
- The combined state of all processes

Distributed Snapshot
- A collection of snapshots, one for each process

Logical clock

Happened-Before Relation (→)

- Captures the behavior of underlying dependencies between the events
- Causality
- Concurrent events

Space-time diagram

Lamport's logical clock

Lamport's time-stamping method

- Events are ordered in a distributed system without the need for physical clocks
- Time-stamping method orders events consisting of transmission of messages
- An event is defined every time a process sends a message: the event corresponds to the time the message leaves the process
- Each system i in the network
  - Maintains a local counter, C_i, which represents the clock for that system
  - When the system transmits a message, it first increments its clock by 1
  - The message sent has the format

$$(m, T_i, i)$$

where

m = contents of the message
T_i = timestamp for this message, set to C_i
i = identifier for this site

Contd..

Lamport's time-stamping method (cont.)

- When the message is received, the receiving system *j* sets its clock to one more than the maximum of its current value and the incoming time-stamp

$$C_j = 1 + max\ [\ C_j,\ T_i\ ]$$

- Ordering of events at every site is determined by the following rule: Message *x* from site *i* proceeds message *y* from site *j* if

1. $T_i < T_j$, or
2. $T_i = T_j$ and $i < j$

- The time associated with each message is the time-stamp of the message

Limitation

Observations

- Ordering obtained with this method does not necessarily correspond to the actual time sequence
- However, all processes involved agree on the ordering imposed on these events
- The local clocks can be incremented for local events also, but the method does not distinguish between those events and the sending of messages
- The method can be used for sequencing events from different processes only if processes exchange messages

Vector clocks

Each process P_i has a clock C_i, which is an integer vector of size 'n' (n = number of processes)

For every event 'a' in P_i, the clock has a value $C_i(a)$, called the time-stamp of event 'a' in P_i

The elements of clock $C_i(a)$ are the clock values of all processes, e.g.

- $C_i[i]$, the i-th entry, is P_i clock value at 'a'
- $C_i[j]$, for $j \neq i$ is P_i's best guess of P_j's logical time (last event in P_j communicated to P_i)

Implementation rules

1. C_i incremented for every event 'a' in P_i

$$C_i[i] \leftarrow C_i[i] + d, \text{ where } d > 0$$

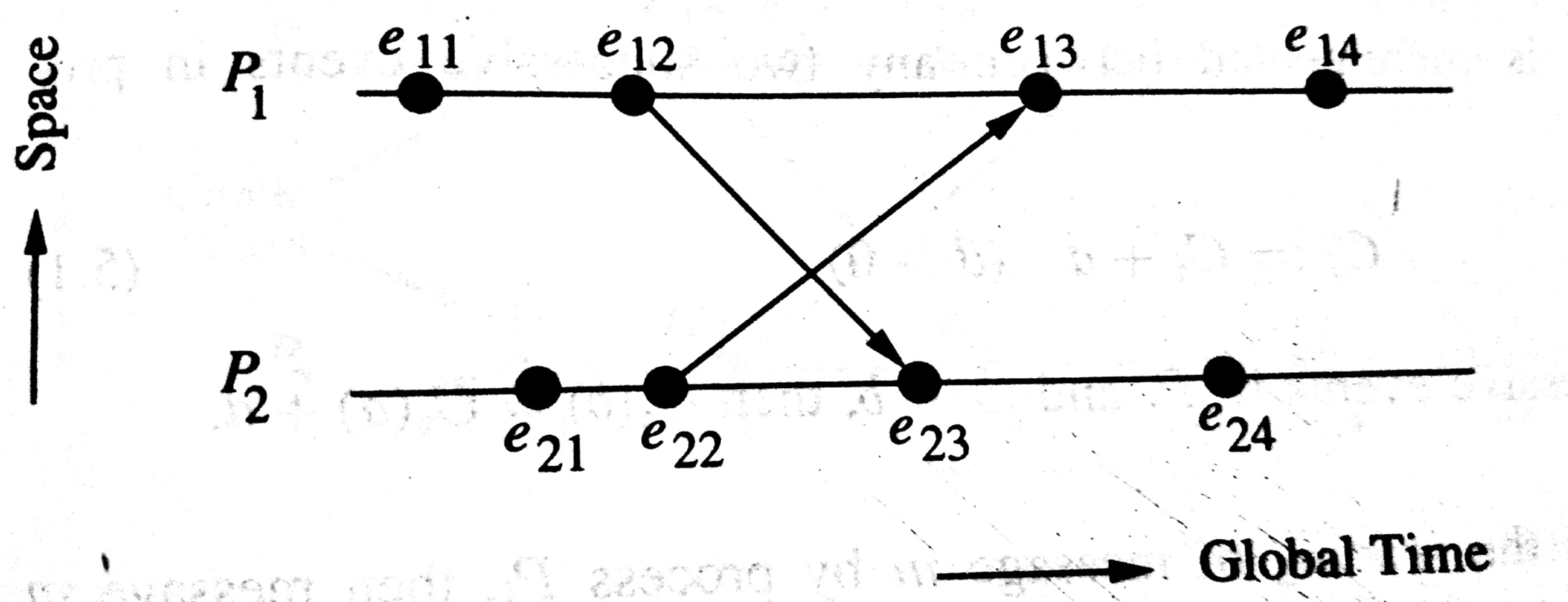
2. If event 'a' is P_i sending message 'm', then message 'm' receives vector time-stamp

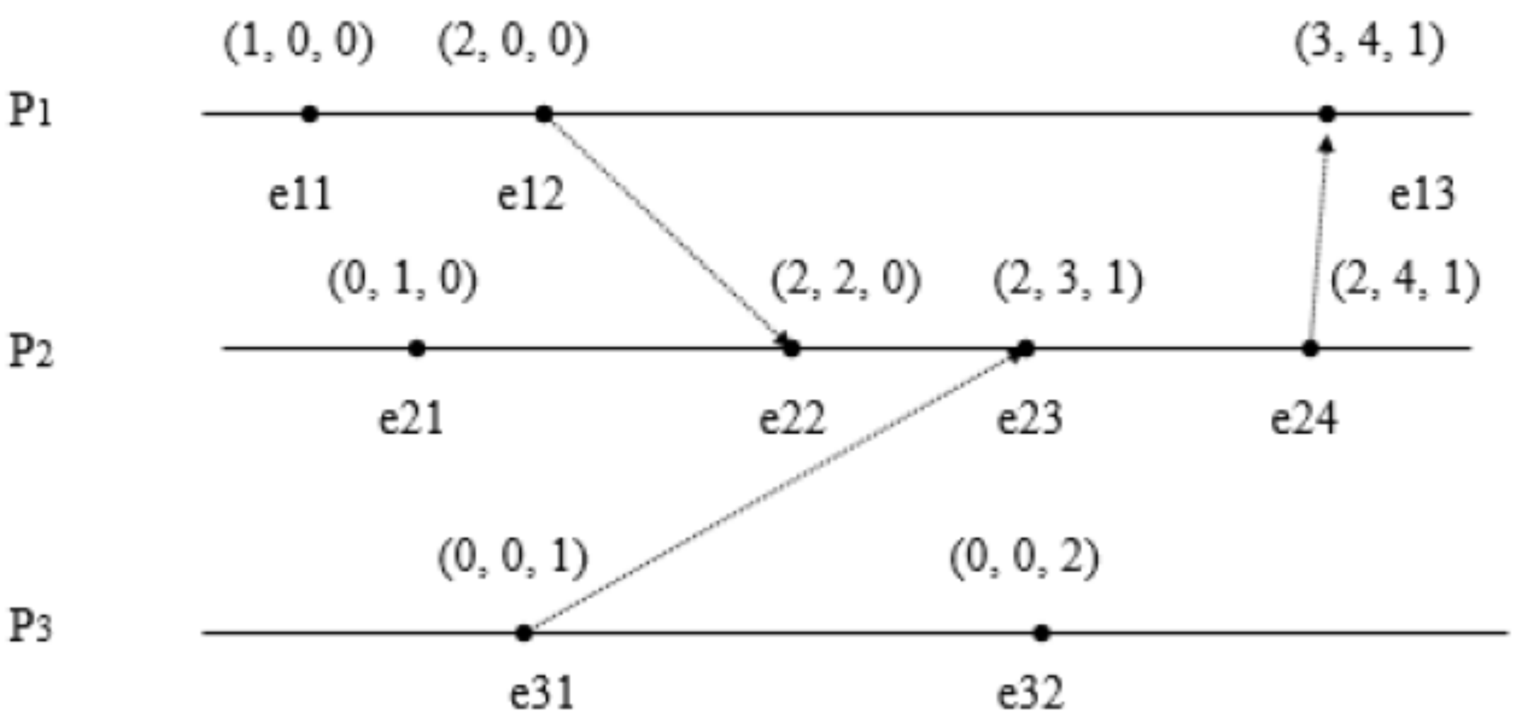
$$t_m = C_i(a)$$

When P_j receives message 'm', its clock C_j updated

$$\forall\, k,\ C_j[k] \leftarrow \max(C_j[k], t_m[k])$$

Logical and vector time




(1, 0, 0)
(2, 0, 0)
(3, 4, 1)
P1
e11
e12
e13
(0, 1, 0)
(2, 2, 0)
(2, 3, 1)
(2, 4, 1)
P2
e21
e22
e23
e24
(0, 0, 1)
(0, 0, 2)
P3
e31
e32

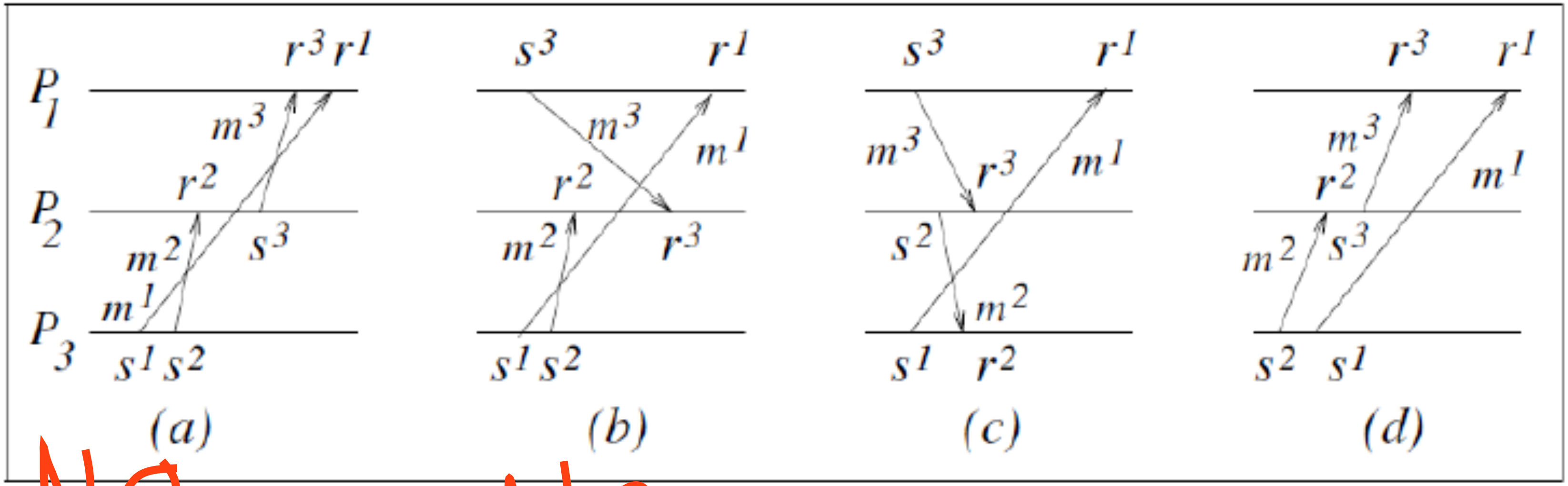
Causal ordering of messages

- Proposed by Birman and Joseph
- Send(M1) → Send(M2)
- Every receipt of M1 must be before receipt of M2
- Replicated database

4 replicated databases: for redundancy and recovery

Database1	R-DB2	R-DB3	R-DB4
Write(a,1)	read(a)	write(a,2)	w(a,1)
Read (a)	write(a,1)	read(a)	w(a,2)
Write(a,2)	write(a,2)	write(a,1)	r(a)

Is causal ordering of messages maintained in the following?



NO

No

Birman- Schiper- Stephenson (BSS) protocol

- Before *broadcasting* m, process P_i increments vector time $VT_{Pi}[i]$ and timestamps 'm'.
- Process $P_j \neq P_i$ receives 'm' with timestamp VT_m from P_i , delays delivery until both:
  - $VT_{Pj}[i] = VT_m[i] - 1$ // received all previous m's
  - $VT_{Pj}[k] >= VT_m[k]$ for every $k \in \{1,2,\ldots, n\} - \{i\}$

  // received all messages also received by P_i before sending message 'm'
- When P_j delivers 'm', VT_{Pj} is updated by IR2 of vector clock

Schiper-Eggli-Sandoz (SES) protocol

- No need for broadcast messages.
- Each process maintains a vector V_P of size N - 1, N being the number of processes in the system.
- V_P is a vector of tuple (P',t): P' the destination process id and t, a vector timestamp.
- T_m: logical time of sending message 'm'
- T_{pi}: present logical time at p_i
- Initially, V_P is empty.

Sending a Message:

- Send message M, time stamped t_m, along with V_P1 to P2.
- Insert (P2, t_m) into V_P1. Overwrite the previous value of (P2,t), if any.
- (P2,t_m) is not sent. Any future message carrying (P2,t_m) in V_P1 cannot be delivered to P2 until $t_m < T_{p2}$.

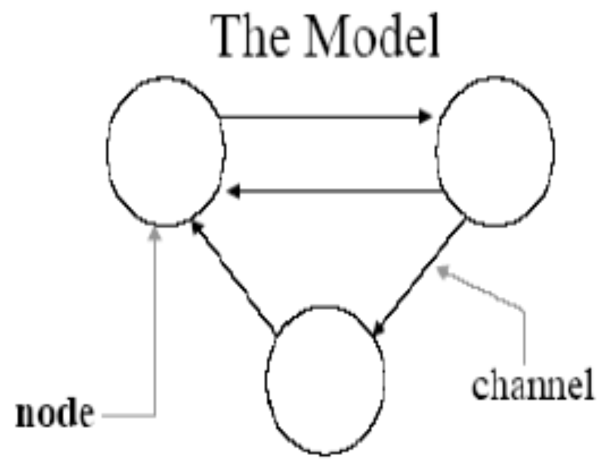
Delivering a message

- If V_M (in the message) does not contain any pair (P2, t), it can be delivered.
- /* (P2, t) exists in V_M*/ If t !< Tp2, buffer the message. (Don't deliver).
- else (t < Tp2) deliver it

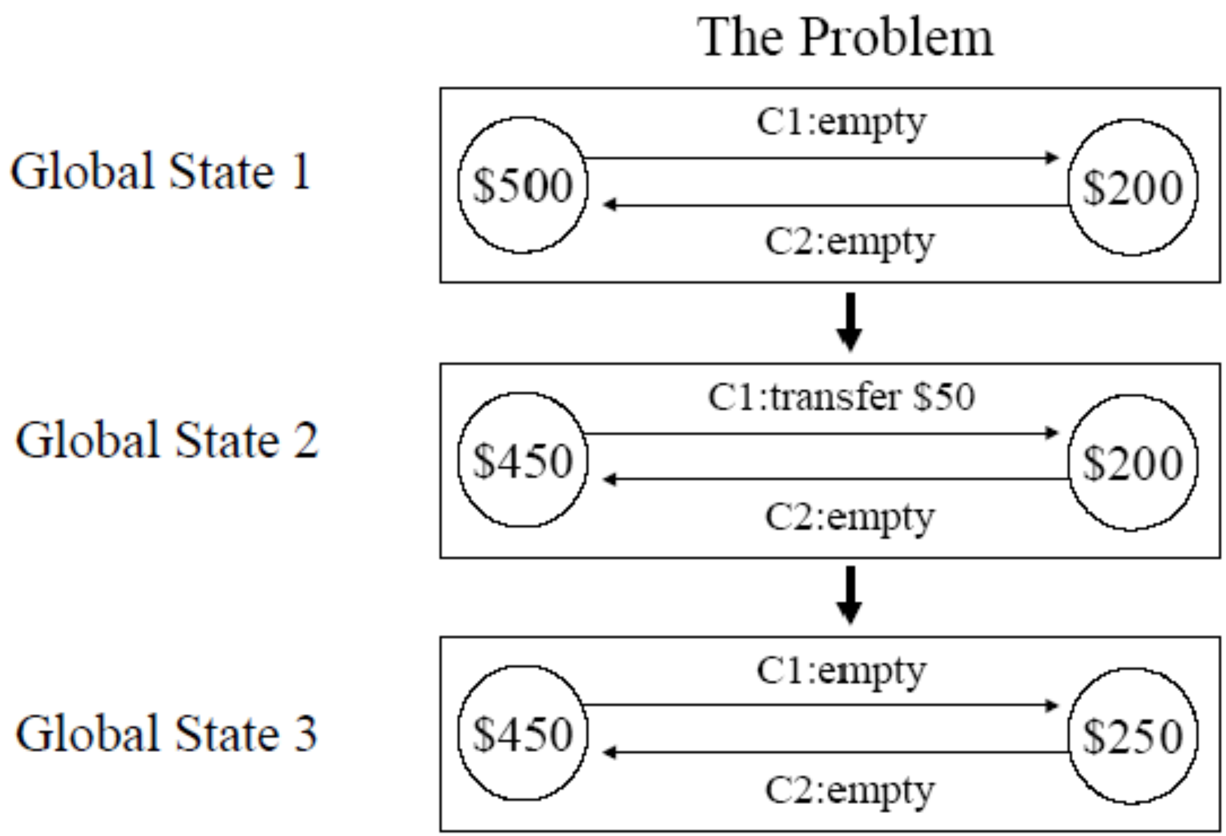
Use IR1 and IR2

Global State : The Model

- Node properties:
  - No shared memory
  - No global clock
- Channel properties:
  - FIFO
  - loss free
  - non-duplicating



Global state



Global state

- n: number of messages sent by A in the channel before A's state was recorded.
- n': number of messages sent by A in the channel before channel's state was recorded.

n>=n'

- m': no. of messages received along channel before B's state recording
- m: no. of messages received along channel by B before channel's state was recorded.

n'>=m

n>=m

Important terms

Transit

Inconsistent

Consistent global state

Strongly consistent global state

Chandy-Lamport Global state recording algorithm

- Problem: record a global snapshot
  - State for each process
  - State for each channel
- System Model:
  - N process in the system
  - Two Unidirectional channels between each ordered process pair
  - Communication in channel is FIFO
  - No failures
  - All messages arrive intact and are not duplicated

Global state

innovate achieve lead

- Snapshot should not interfere with normal application actions
- Should not require applications to stop sending messages
- Each process is able to record its own state
    - Application defined state
    - Heap, register, stack, program counter, code, (coredump)
- Global state is collected in distributed manner
- Any process may initiate the snapshot

Chandy-Lamport Global state recording algorithm

- **First, Initiator P*i* records its own state**
- **Initiator process creates special messages called "Marker" messages**
  - Not an application message, does not interfere with application messages

- **for *j=1 to N* except *i***
  P*i* sends out a Marker message on outgoing channel C_{ij}
  - (N-1) channels
- Starts recording the incoming messages on each of the incoming channels at P*i*: C_{ji} (for $j=1$ *to* N except i)

Chandy-Lamport Global state recording algorithm

Whenever a process P*i* receives a Marker message on an incoming channel C_{ki}

- **if** (this is the first Marker P*i* is seeing)
  - P*i* records its own state first
  - Marks the state of channel C_{ki} as "empty"
  - For *j=1 to N* except *i*
    - P*i* sends out a Marker message on outgoing channel C_{ij}
  - Starts recording the incoming messages on each of the incoming channels at P*i*: C_{ji} (for *j=1 to N* except *i* and *k*)
- **else // already seen a Marker message**
  - Mark the state of channel C_{ki} as all the messages that have arrived on it since recording was turned on for C_{ki}

Chandy-Lamport Global state recording algorithm

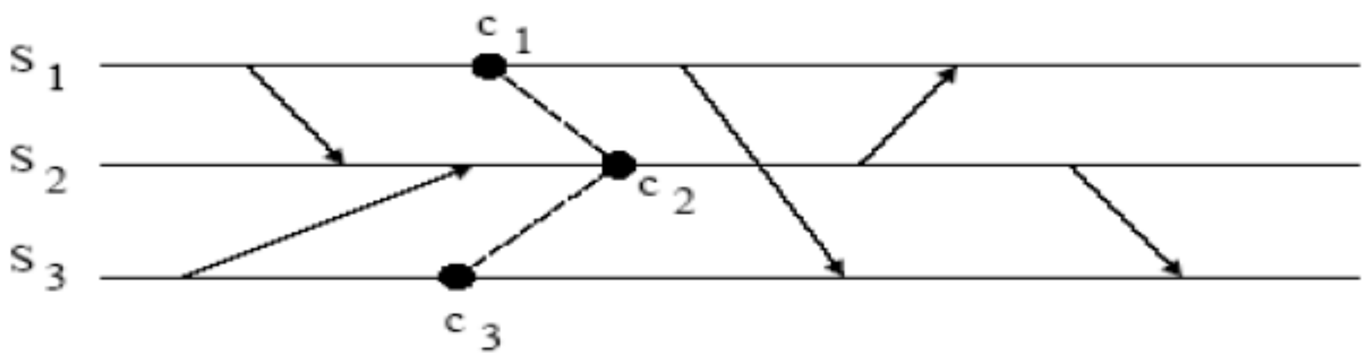
The algorithm terminates when

- All processes have received a Marker
  - To record their own state
- All processes have received a Marker on all the (*N-1*) incoming channels at each
  - To record the state of all channels

Then, (if needed), a central server collects all these partial state pieces to obtain the full global snapshot

Cuts

- A cut is a set of cut events, one per node, each of which captures the state of the node on which it occurs.
- It is also a graphical representation of a global state.



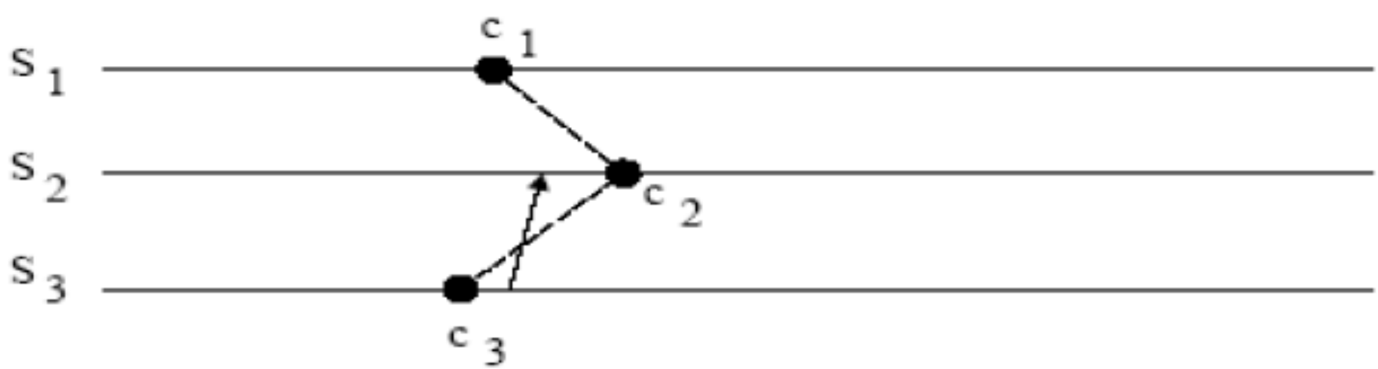
$C = \{ c_1, c_2, c_3 \}$

Consistent Cut

A cut C = {c1, c2, c3, … } is consistent if for all sites there are no events ei and ej such that:

(ei --> ej) and (ej --> cj) and (ei -/-> ci), c_i, c_j Є C

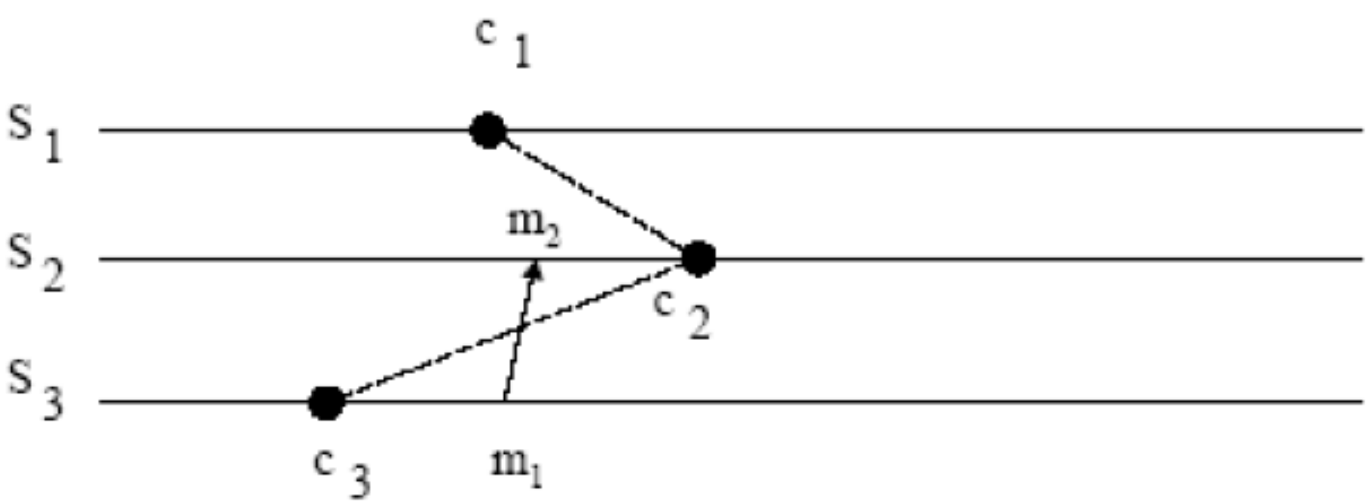
An inconsistent cut :



Ordering of Cut Events

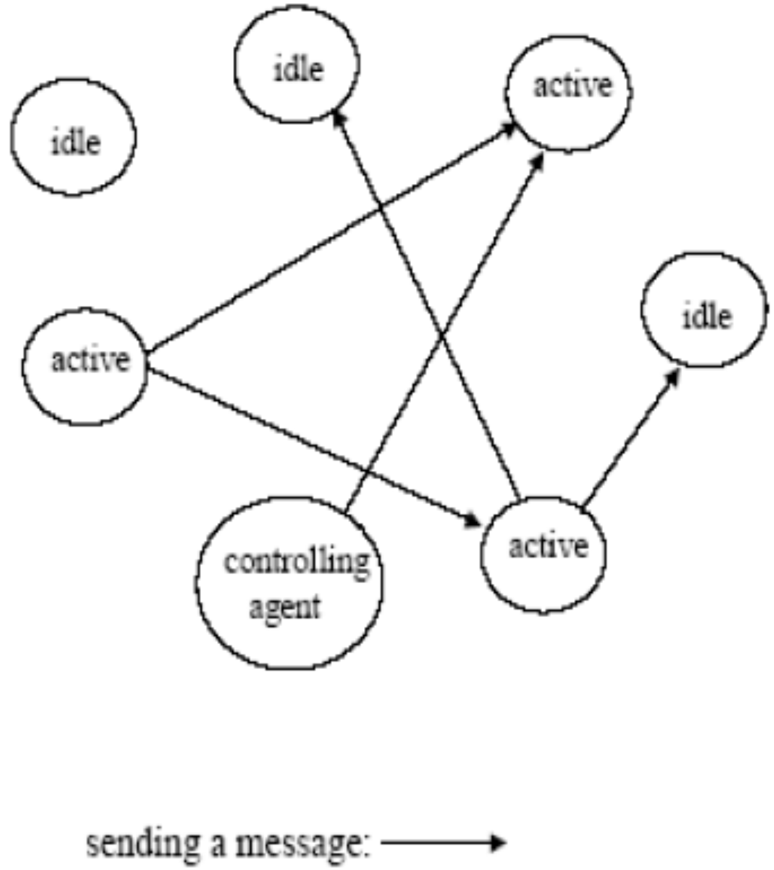
The cut events in a consistent cut are not causally related. Thus, the cut is a set of concurrent events and a set of concurrent events is a cut.

Note, in this inconsistent cut, c3 --> c2.



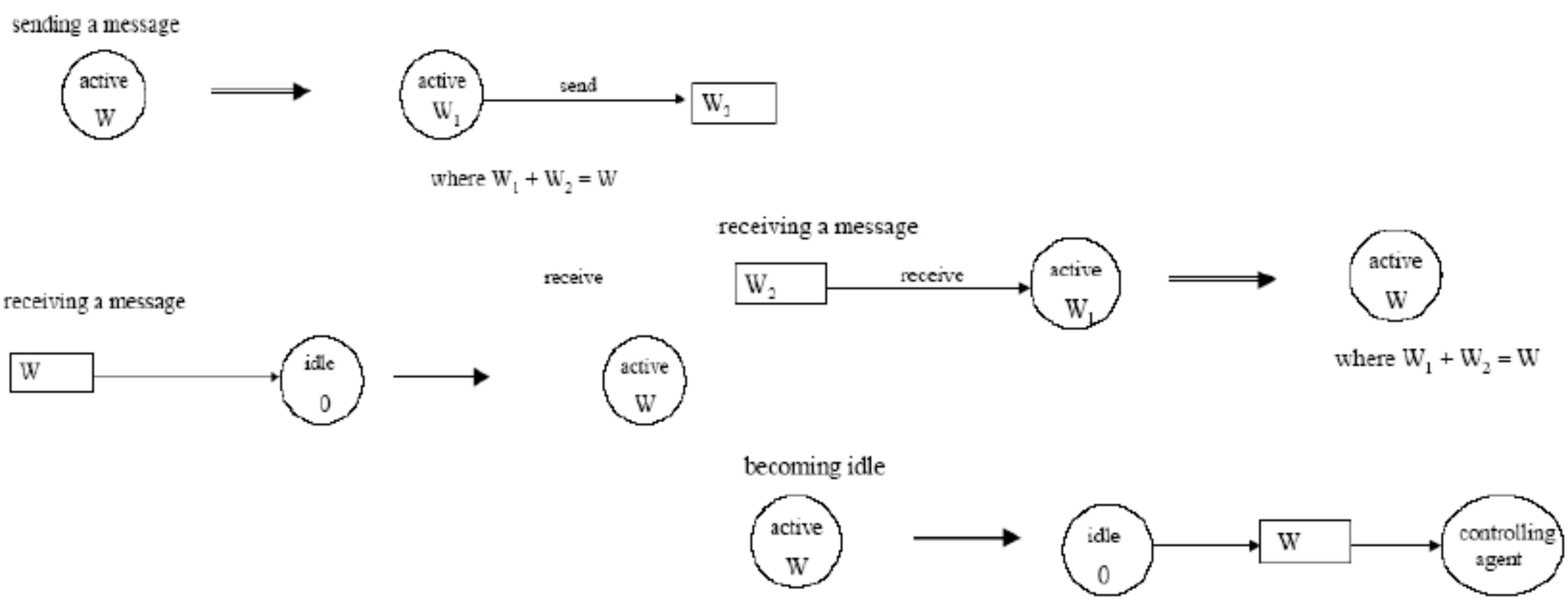
Termination Detection

In a distributed computation, when are all of the processes become
idle (i.e., when has the computation terminated)?



Huang's Algorithm

The computation starts when the controlling agent sends the first message and terminates when all processes are



Any questions