Distributed Systems
CS 380D

Vijay Chidambaram

Spring 2020

Types of knowledge

- Common knowledge:
    - known by everyone in group
    - each node **can** assume others know this
- Distributed knowledge:
    - known by some members of group
    - a node **cannot** assume others know this
- Simultaneous actions requires common knowledge

Common Knowledge

- Impossible to obtain if communication is over unreliable channels
- Demonstrated in the Coordinated Attack Problem
- Internal Common Knowledge:
  - assume something is common knowledge
  - hope no node encounters state that disproves assumption

Muddy Children Puzzle

- n children play, k get muddy
- Each can observe all others, don't know their own state
- Dad says "at least one of you is muddy"
- Dad asks each of them: "do you know if you are muddy"?
- claim: After k-1 rounds, all children will answer yes

Muddy Children Puzzle

- Children get information from:
  - Observation of other children
  - Hearing what other children say
  - Inferences based on previous rounds
- Common knowledge: father says at start “at least one of you is muddy”

Proof by induction

- k = 1:
  - Muddy child observes all others are clean
  - But father said someone is muddy
  - Hence child realizes they are muddy, answers yes
  - Once other children hear muddy child answer yes, they also answer yes
- k = 2:
  - Each muddy child observes one other muddy child
  - in first round, k = 1, all answer no as they are unsure of their own state
  - Muddy child realizes they are muddy, since other muddy child answered no in first round (hence other child must see someone muddy)
  - In second round, all answer "yes"

Proof by induction

- k = 3
  - Say muddy children are a, b, c
  - if a is clean, b and c would have answered yes in second round
  - Hence a is not clean; b and c do similar reasoning
  - All answer yes on third round

Does father need to provide common knowledge?

- One might think no: for $k > 1$, seems like children get the information from direct observation
  - However, it is not common knowledge
  - For $k = 2$, muddy child a observes muddy child b. But does not know if b observes a, and therefore knows $k >= 1$

Does father need to provide common knowledge?

- Showing it does not work for k = 2:
  - Muddy children are A and B
  - In first round, even if A had seen all clean kids, they would have still answered “no” (because they do not know k >= 1)
  - In second round, A and B realizing they are muddy depends on muddy child saying yes in round 1
  - A saying “no” in round 1 does not provide B with any information
  - B still thinks k = 1 or k = 2

Does father need to provide common knowledge?

- Valid sequence if k = 1 from B's viewpoint:
  - A is only muddy child
  - A does not realize k > = 1, cannot decide between k = 0 and k = 1
  - A says "no" in first round
- B still cannot decide between k = 1 or k = 2 (both can happen with prior seq)

Common knowledge

- k >= 1 is distributed knowledge, not common knowledge
- This case clearly shows the difference between the two

Hierarchy of States of Knowledge

- Agent's knowledge depends on:
  - Starting knowledge
  - Observed history since start
- If agent i knows P then $K_i(P)$
- Agents know only true things

Hierarchy of States of Knowledge

- D(G, P) = group G has distributed knowledge of P (union of knowledge of G members = P)
- S(G, P) = someone in G knows P
- E(G, P) = everyone in G knows P
- E(G, K, P) = E(E(E.. E(G,P))))) k times
- E(E(G,P)) = everyone in G knows that everyone in G knows P
- Common knowledge: E(G, K, P) for all K >= 1

Muddy Children Puzzle

- m = "at least one child is muddy"
- Without father speaking,
  - E(G, K–1, m) = true
  - E(G, K+1, m) = false

Muddy Children Puzzle

- E(G, m) = true
- In k = 2,
  - E(G, 1, m) = true, everyone knows m
  - E(G, 2, m) = false, everyone does not know everyone knows m
  - Specifically, with muddy children A and B, A does not know B knows K >= 1
- Fathers statement makes E(G, 2, m) = true

Knowledge in distributed systems

- Communication in a distributed systems seeks to move up the hierarchy of knowledge:
  - changing S(G, P) = E(G, P) = C(G, P)
- Fact discovery:
  - Changing D to S to E to C
  - Example: finding deadlock in a set of distributed locks
- Fact publication:
  - Changing S to C
  - Example: new protocol for communication

Common Knowledge

- How does one establish it?
  - By being part of a community
    - Membership procedure imparts common knowledge
    - Example: community of licensed drivers knows what signs mean
  - By being co-present at knowledge creation
    - Example: children being in same room as father when he makes announcement

Coordinated Attack Problem

- General A sends time in message to General B
- A will not attack without ack from B
- B sends ack to A
- But B will not attack without ack from A
- A sends Ack(Ack(A)) to B
- A will not attack without ack of this message
- And so it goes.. A and B cannot agree with finite messages
- Can use induction to prove no set of K messages is enough

Coordinated Attack Problem

- Generals A and B need common knowledge of the attack time
- After A sends the first message to B, $E(G)$ = true, but $E(E(G))$ is not
- After A sends the ack to B's ack, $E(E(G))$ = true, but $E(3, G)$ = false
- Coordinated attack requires $C(G) = E(k, G)$ = true for any k

Jan 28
More on Common Knowledge

Vijay Chidambaram

Knowledge in Processors

- Ground facts are facts about the state of the system: represents the raw state without semantics
- At each point in the protocol, a processor has a view
- A processor knows all the facts that follow from its view at a given point

View-based interpretation

- Every node has a view based on its history
- Every view is associated with a set of facts both directly known and inferred

Coordinated Attack Problem

- Even with guaranteed delivery, if messages can be delayed an unbounded amount of time, attack cannot be coordinated
- Why? No guarantee that other party seems message before attack time

Coordinated Attack Problem

- What if the delay in delivery time was bounded to “e”?
- Still not possible to coordinate attack
- Lets say Y receives a message from X at time TD
- Y knows X will not assume Y has seen it until e time has passed (TS + e)
- But TS could also be TD, message could be delivered instantly
- Y has to wait until TD + e to be sure that X knows Y has received the message
- So in total, 2e time units has to pass until it is common knowledge among X and Y that X sent a message to Y
- With each round, the time units keep increasing: k*e for K rounds
- Since common knowledge requires arbitrary K to hold, it follows that an infinite amount of time has to pass

Attaining common knowledge

- Common knowledge is attainable if multiple nodes in the system can **simultaneously** converge on a single option (among many options)
- When one node believes M, all nodes must simultaneously believe M if M is common knowledge
- The histories of all nodes must simultaneously change to reflect M

Common Knowledge in Practice

- Common knowledge needed for simultaneous coordination in a distributed system
- For other types of coordination, weaker states of knowledge is enough
- E-common knowledge: when every agent knows M within time units E
- E-common knowledge achieved through **synchronous broadcast**: all agents guaranteed to receive it within E time units

Stable properties

- E-common knowledge is useful as it allows us to identify stable properties
- A stable property S is a property of the system such that once S becomes true, it is always true
- For example, once the system is deadlocked, it is always deadlock pending some external action

Eventual Common Knowledge

- What to do for async broadcast?
- Eventual common knowledge (M): Every node knows every other node knows M or will know M in the future
- Useful in real-world scenarios: for example, in Byzantine agreement, once a value is agreed upon by one processor, all other processors decided on this value eventually

Jan 30

Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms

Vijay Chidambaram

Global State of a distributed system

- Union of local state of nodes
- No way to get instantaneous snapshot of all nodes
- Only way to get local state of a node is by sending it a message
- How to find a meaningful global state of the system?

Why is this hard?

- Messages could be dropped or delayed
- Computed Global state may be:
  - Obsolete: state changed since we have checked
  - Incomplete: state of some nodes may be missing
  - Inconsistent: imagine a token is sent from A to B. Computed Global state may show token in both A and B

Computing Global State

- What if we simply used a lot of messages to nodes to compute global state?
- Would help with ensuring global state is complete or current (not obsolete), but will not help with ensuring global state is consistent
- Consistency cannot be achieved by throwing more messages at the problem

Global Predicate Evaluation (GPE)

- We construct a predicate based on the global state
- We want to evaluate whether this global predicate is true or false
- Examples: the system is deadlocked, a majority of nodes are alive and responsive

Modeling the system

- N sequential processes p1.. pn
- Each pair of processes has a channel
- Channels are reliable but may deliver messages out of order
- Why do we model system as async?
  - Physical delays are bounded
  - Software creates unbounded delays

Distributed Computation

- Activity distributed among N processes
- Each process sees three kinds of events:
  - Events local to that process
  - Send message to process pi
  - Receive message from process pj
- All events are recorded in the local history of the process
- Global history is union of histories of all participating processes

Global History

- Global history does not order all events
- An event A is only ordered with respect to event B if A happening affects B in some way
- Events are ordered using "happens-before" relationships

Lamport Clocks

- Notation: e(i, k) = kth event in process i
- e(i, j) < e(i, k) if j < k
- if e(i, j) = send(m), and e(k, l) = receive(m)
  - then e(i, j) < e(k, l)
- if e(i, j) < e(k, l) and e(k, l) < e(m, n)
  - then e(i, j) < e(m, n)
- All other events are considered concurrent
  - consider e(i,j) and e(k,l) concurrent
  - e(i, j) < e(k,l) is false
  - e(k, l) < e(i,j) is also false

Distributed Computation

- Formally, a distributed computation is a partially ordered set (poset) defined by the pair (H, ->)
- Not all events are ordered
- Events inside each process are totally ordered
- Events across processes are partially ordered

Cuts

- A cut of a distributed computation is a subset C of its global history H and contains an initial prefix of each of the local histories
- A cut can be defined by tuple (c1, c2, ..cN)
  - Process pi's last event in the cut is ci
  - P1's last event in the cut is c1
- Frontier of the cut: the set of events e(i, ci) for i=1..n (the last events included in the cut for each process)

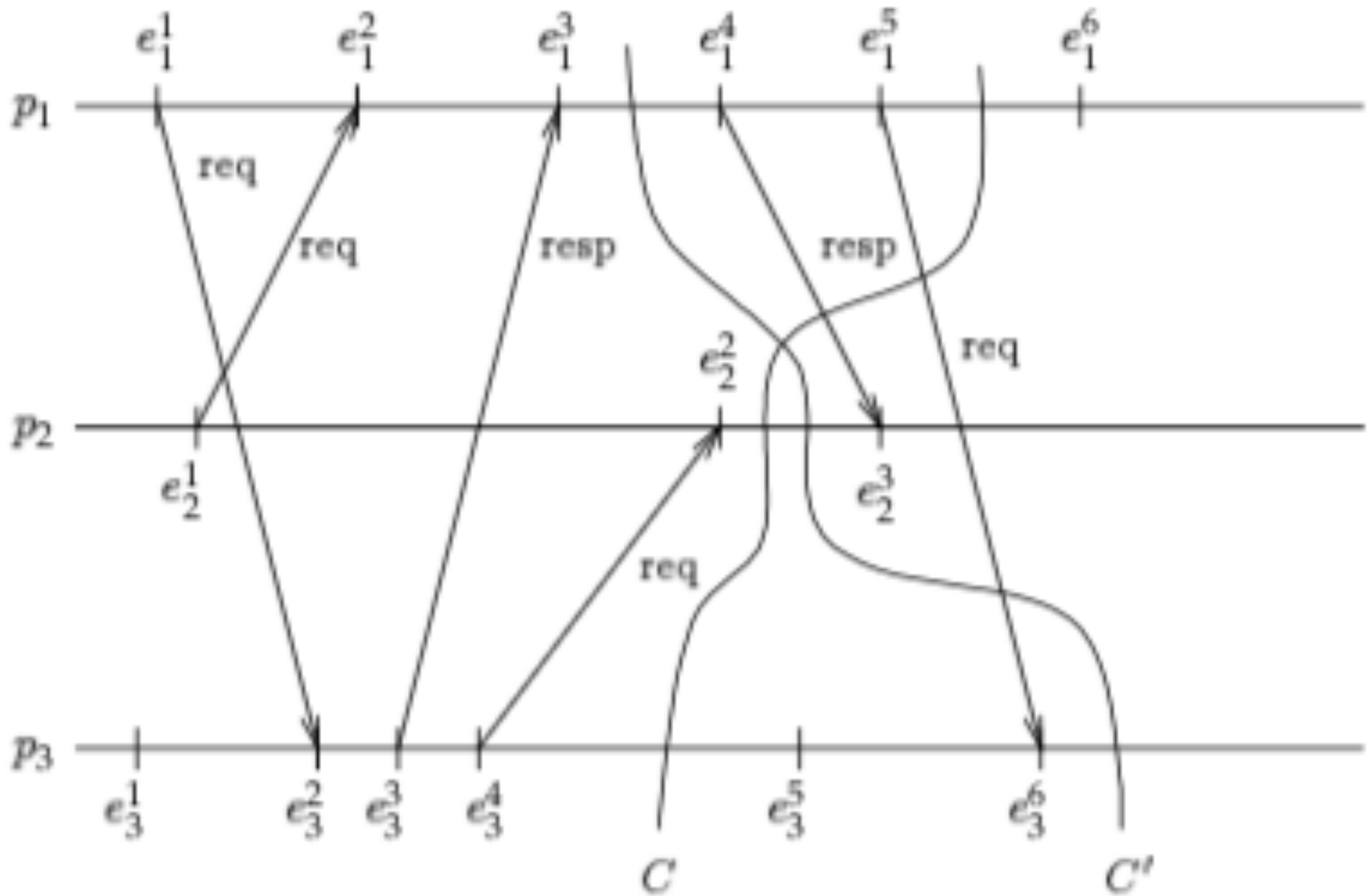


Figure 2. Cuts of a Distributed Computation

Runs

- A run of a distributed computation is total ordering R that includes all of the events in the global history and that is consistent with each local history.
- For process Pi, the events of Pi occur in the same order in R as in the history of Pi
- There are many possible runs for a single distributed computation with history H

Inconsistent Cuts

- Not all cuts are consistent
- If a cut includes receipt of a message but not the sending of the message, it is inconsistent
- More precisely, if $e(i, j) < e(k, l)$ and $e(k, l)$ is in the cut, then $e(i,j)$ must also be in the cut
- Global properties must be checked using consistent cuts (which lead to consistent global states)
- Using inconsistent cuts may lead to determination of “ghost deadlock”

Reachable States

- A run R is said to be consistent if for all events, e1 < e2 implies that e1 appears before e2 in R
- Each (consistent) global state Si of the run is obtained from the previous state Si-1 by some process executing the single event ei
- Si-1 leads to Si
- Sj is reachable from Si, if there is a series of consistent states from Si to Sj such as Si -> Sk -> Sj

Lattice

- The set of all consistent global states of a computation along with the leads-to relation defines a lattice.
- The lattice consists of n orthogonal axes, with one axis for each process
- Each path down the lattice is one run of the distributed system

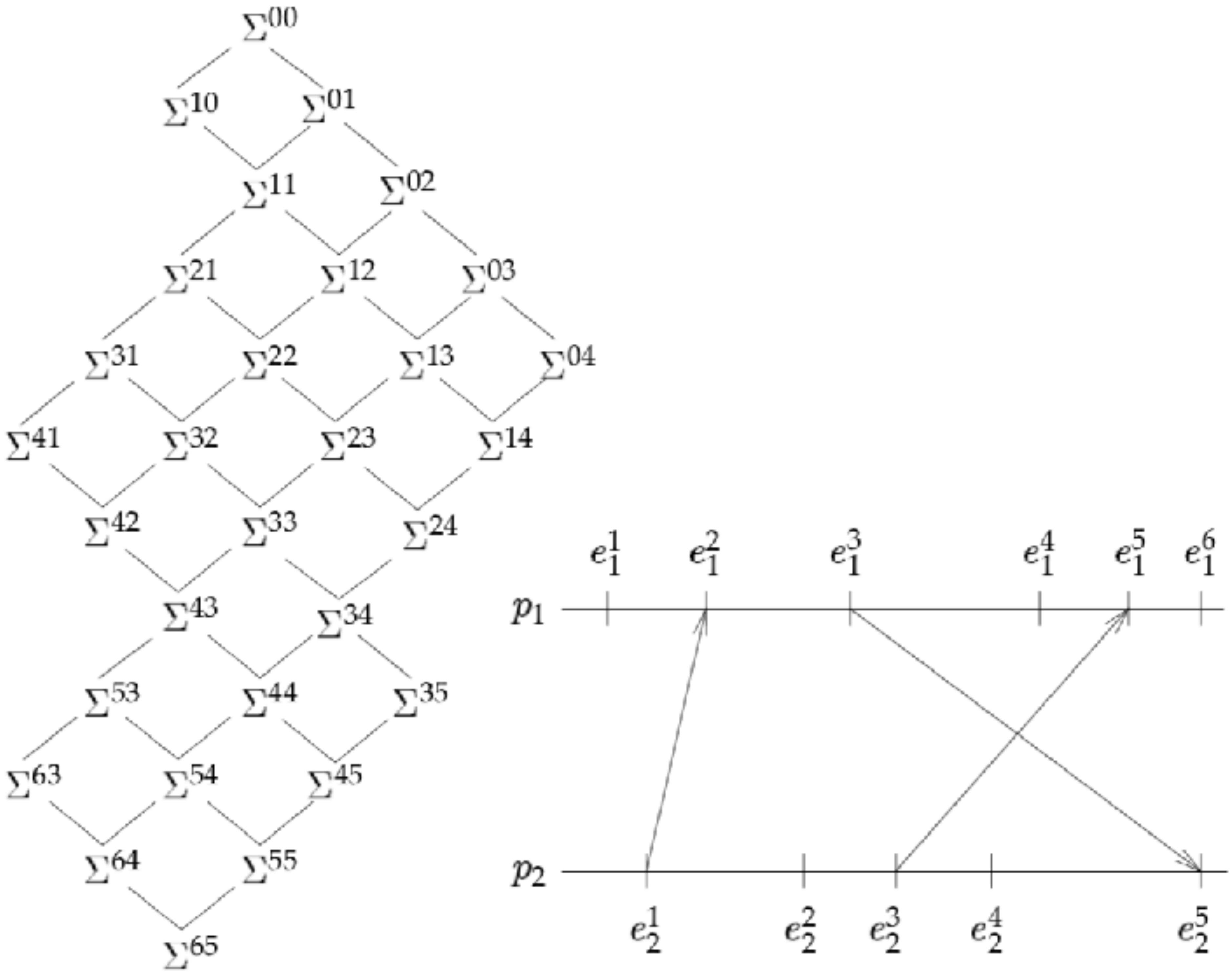


Figure 3. A Distributed Computation and the Lattice of its Global States

Observing a distributed system

- Idea 1: Active Observer
  - One monitor process sends messages to all processes
  - Constructs global state based on responses
  - Can lead to inconsistent cut

Observing a distributed system

- Idea 2: Passive Observer
  - One monitor process gets copy of all messages sent by processes
  - Different monitors observe different cuts (and hence different global states)
  - Consistent observation leads to a consistent run

Observing a distributed system

- We ensure messages from the same process are delivered in order (FIFO delivery)
  - Implemented using per-process sequence numbers
- Delivery Rule:
  - if e1 < e2, then ts(e1) < ts(e2).
  - Deliver messages in timestamp order

Feb 4
Consistent Global States
(Part 2)

Vijay Chidambaram

Consistent observations

- A consistent observation is one that corresponds to a consistent run.
- It is the possibility of messages being reordered by channels that leads to undesirable observations
- We can restore order to messages between pairs of processes by defining a delivery rule for deciding when received messages are to be presented to the application process.

FIFO Delivery

- FIFO Delivery: sendi(m) -> sendi(m') => deliveri(m) -> deliveri(m')
- Clock Condition: e -> e' => Timestamp(e) < timestamp(e')
- Delivery Rule DR1: At time t, deliver all received messages with timestamps up to t - delta in increasing timestamp order

Gap Detection

- Given two events e and e′ along with their clock values LC(e) and LC(e′) where LC(e) < LC(e′) , determine whether some other event e″ exists such that LC(e) < LC(e″) < LC(e′)
- A message m received by process p is called stable if no future messages with timestamps smaller than TS(m) can be received by p.
- DR2: Deliver all received messages that are stable at p0 in increasing timestamp order.

Causal Delivery

- FIFO delivery is per-process
- Causal Delivery extends it across processes
- Causal Delivery: sendi(m) -> sendj(m') => deliveri(m) -> deliverj(m')
  - Note i and j can be different
- FIFO delivery between all pairs of processes not enough for causal delivery
- A -> B, A -> C; B->C
  - sendAB(M1) -> sendAC(M2)
  - B gets M1, sends it to C
  - C gets M2 before M1.
  - FIFO delivery is ensured, causal delivery is not.

Consistent Observations

- If the observer process uses a delivery rule that satisfies Causal Delivery, then all its observations will be consistent
- How do we build this in a practical distributed system?

Building the Observer

- Assume Observer has two parts: the Logic Controller and the Network Controller
- The Network Controller gets events sent by other nodes, decides in what order to show them to Logic Controller
  - Delivery rules are implemented here
- Logic Controller takes actions based on what it sees (“declare deadlock”)

Building the Observer

- Network Controller gets two events E1 and E2
- Timestamp = TS
- TS(E1) < TS(E2)
- This doesn't mean E1 < E2
- Only this is true: E1 < E2 => TS(E1) < TS(E2)
- So we know E2 < E1 is false, E1 and E2 could still be concurrent

Strong Clock Condition

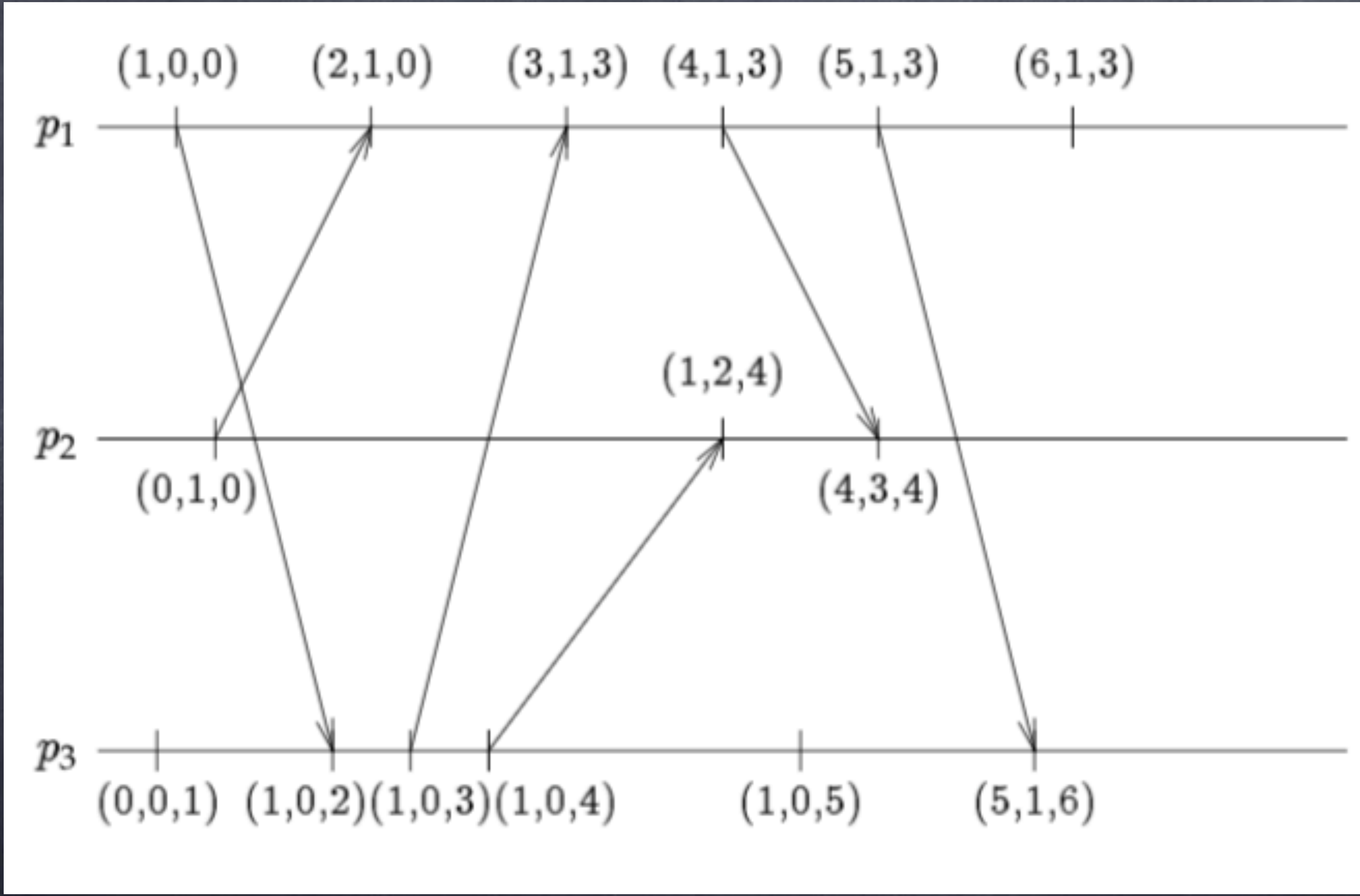
- Need stronger condition to implement Network Controller
- Strong Clock Condition:
  - E1 < E2 => TS(E1) < TS(E2)
  - TS(E1) < TS(E2) => E1 < E2

Implementing the Strong Clock Condition

- Brute force approach:
  - At each node, keep the Causal History C
  - Causal History C(e) is the set of all events e′ such that e′ < e
  - All previous local events at node are part of C(e)
  - When A sends a message to B, C(receipt of message M) = C(previous local event at B) union C(sending event at A)
  - E1 < E2 if C(E1) is a subset of C(E2)
- Problem: the set C grows too large too quickly, impractical to maintain

Vector Clocks

- Each node maintains a vector V[n] where there are n nodes
- V = 0 on initialization for all nodes
- For local event Ei at node i, update V[i] = V[i] + 1
- On receipt(M),
  - V = max(V, vector-TS(M)) for each element of V
  - V[i] +=1
- V[j] = number of events of j that casually precede this event (when this process is not j)
- V[i] = number of local events when this process is i


(1,0,0)
(2,1,0)
(3,1,3)
(4,1,3)
(5,1,3)
(6,1,3)
p1
(1,2,4)
p2
(0,1,0)
(4,3,4)
p3
(0,0,1)
(1,0,2)
(1,0,3)
(1,0,4)
(1,0,5)
(5,1,6)

Using Vector Clocks

- V1 < V2 if no element of V2 is bigger than its corresponding element in V1
- Strong Clock Condition: E1 < E2 <=> V(E1) < V(E2)
- V1 and V2 are concurrent if V1[i] > V2[i] for some i and V2[j] > V1[j] for some j

Using Vector Clocks in the Network Observer

- All events sent to observer have vector timestamps
- Network Observer obtains set of events M, then decides on order to deliver them
- A message can be delivered as soon as Network Observer determines there are no events that casually precede this message
- Consider message M from J
- M′ is last message delivered from K (K != J)
- To deliver M, NO has to determine that:
  - There is no earlier message from J that is undelivered
    - if V(M)[J] -1 message have already been delivered, there cannot be an earlier message
  - There is no undelivered message M″ such that sendK(M′) < send (M″) < sendJ(M)
    - V(M′)[K] >= V(M)[K] for all K => V(M) < V(M′)

Using Vector Clocks in the Network Observer

- Network Observer maintains array D, initialized to 0
- Deliver message M with time stamp V from J as soon as:
  - D[J] = V[J] - 1
  - D[K] >= V[K], for all K != J
- When Network Observer delivers M, D is updated by setting D[j] = V[j]

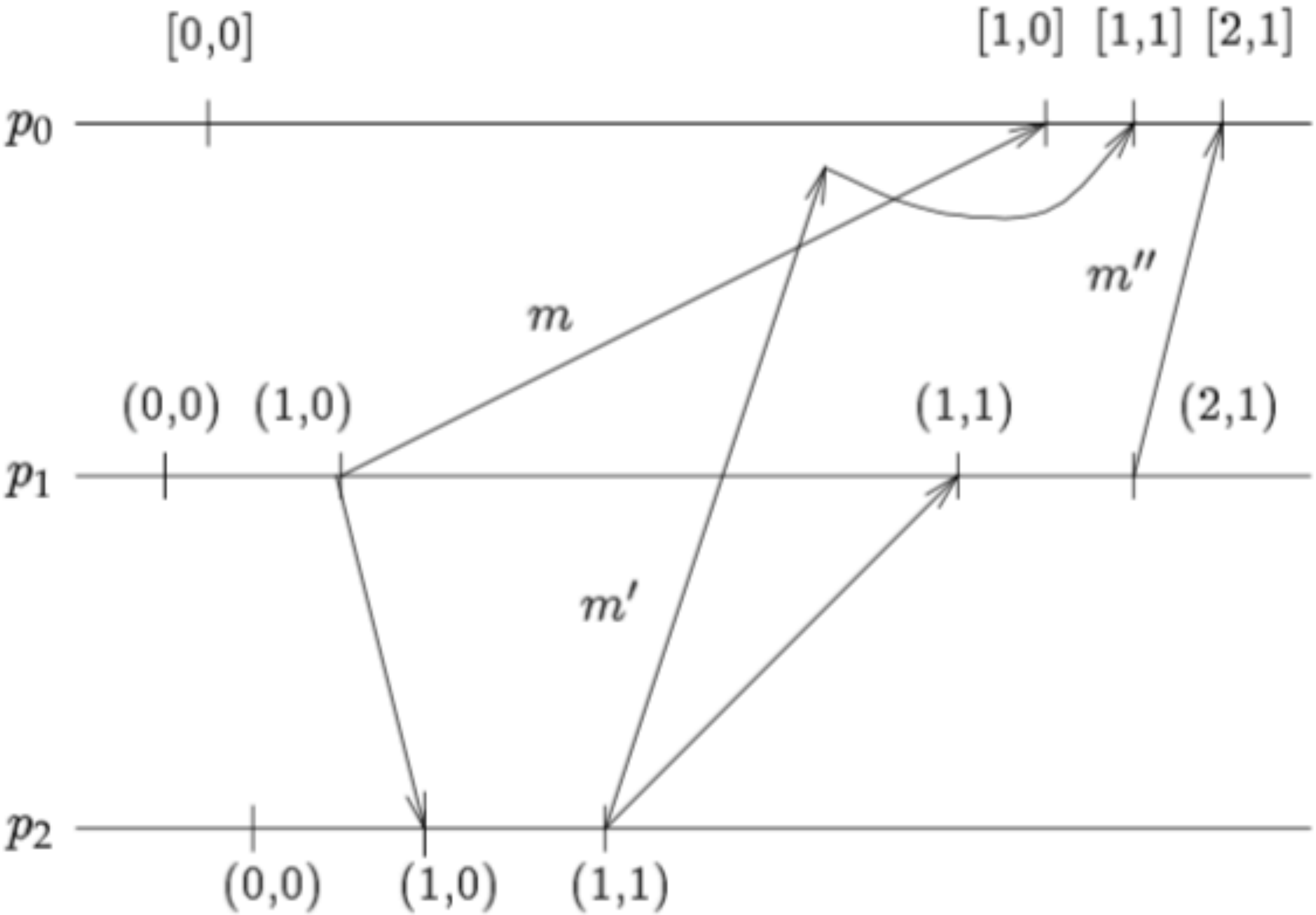


Figure 8. Causal Delivery Using Vector Clocks

Feb 6

Distributed Snapshots

Vijay Chidambaram

Using Vector Clocks in the Network Observer

- Network Observer maintains array D, initialized to 0
- Deliver message M with time stamp V from J as soon as:
  - D[J] = V[J] - 1
  - D[K] >= V[K], for all K != J
- When Network Observer delivers M, D is updated by setting D[j] = V[j]

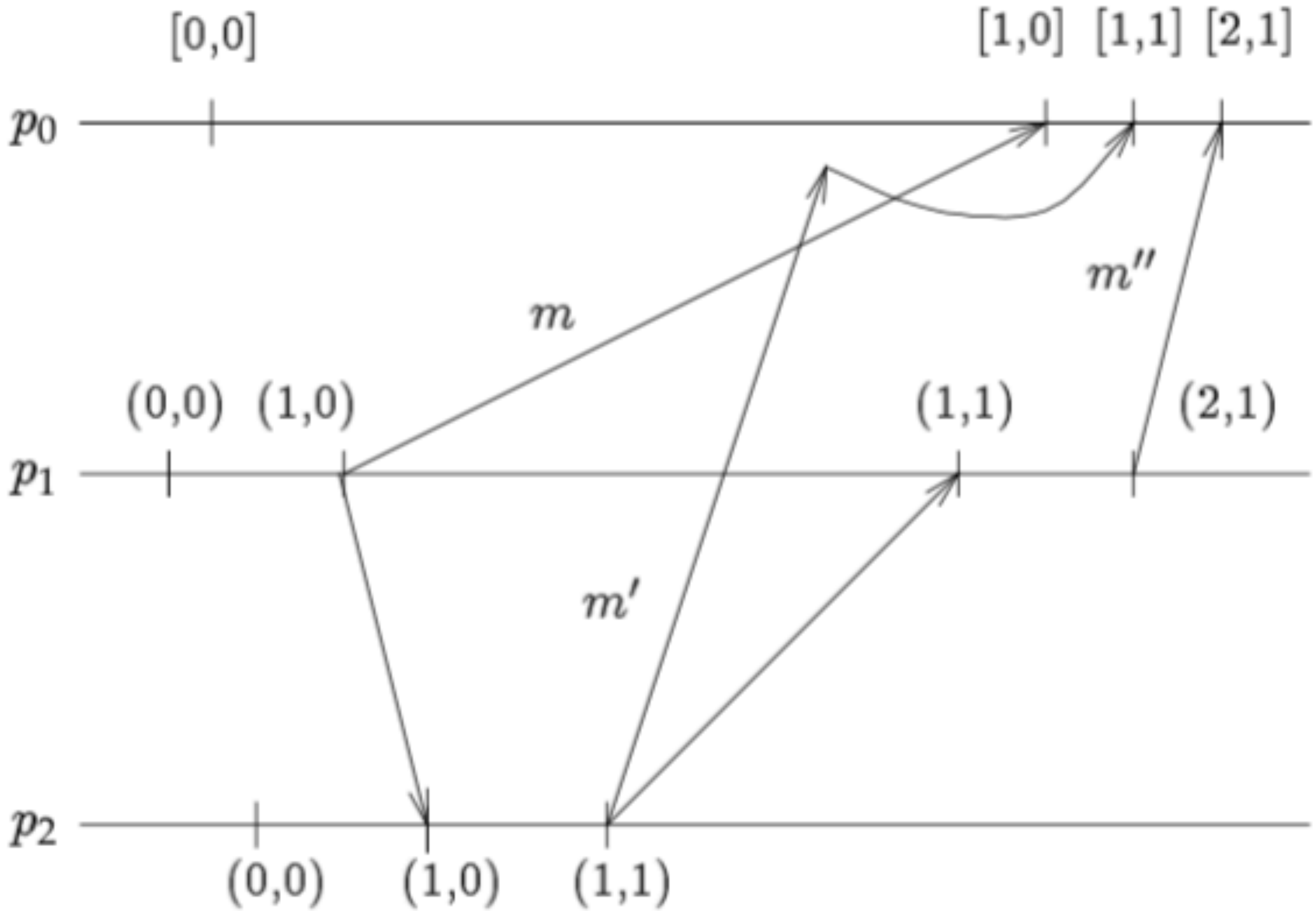


Figure 8. Causal Delivery Using Vector Clocks

State Machines

- Each process in a distributed system is modeled as a state machine
- The process is in an initial state I
- Upon getting a message M from another process, the process transitions to another state S1
- In state S1, process responds to other messages by moving to other states S2..SN
- Processes transition only on receiving messages

Distributed Snapshot Algorithm

- How to compose a global snapshot based on the snapshot of individual nodes?
- How to deal with double-counting or missing state because of messages that were in flight?

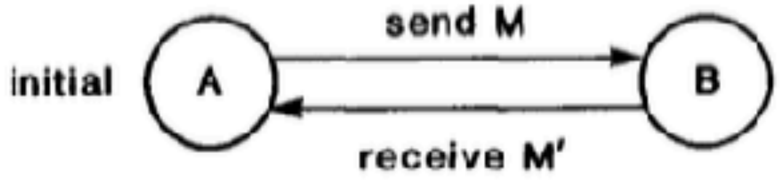


Fig. 5. State-transition diagram for process p in Example 2.2.

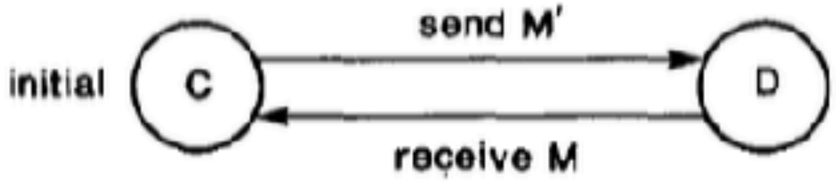


Fig. 6. State-transition diagram for process q in Example 2.2.

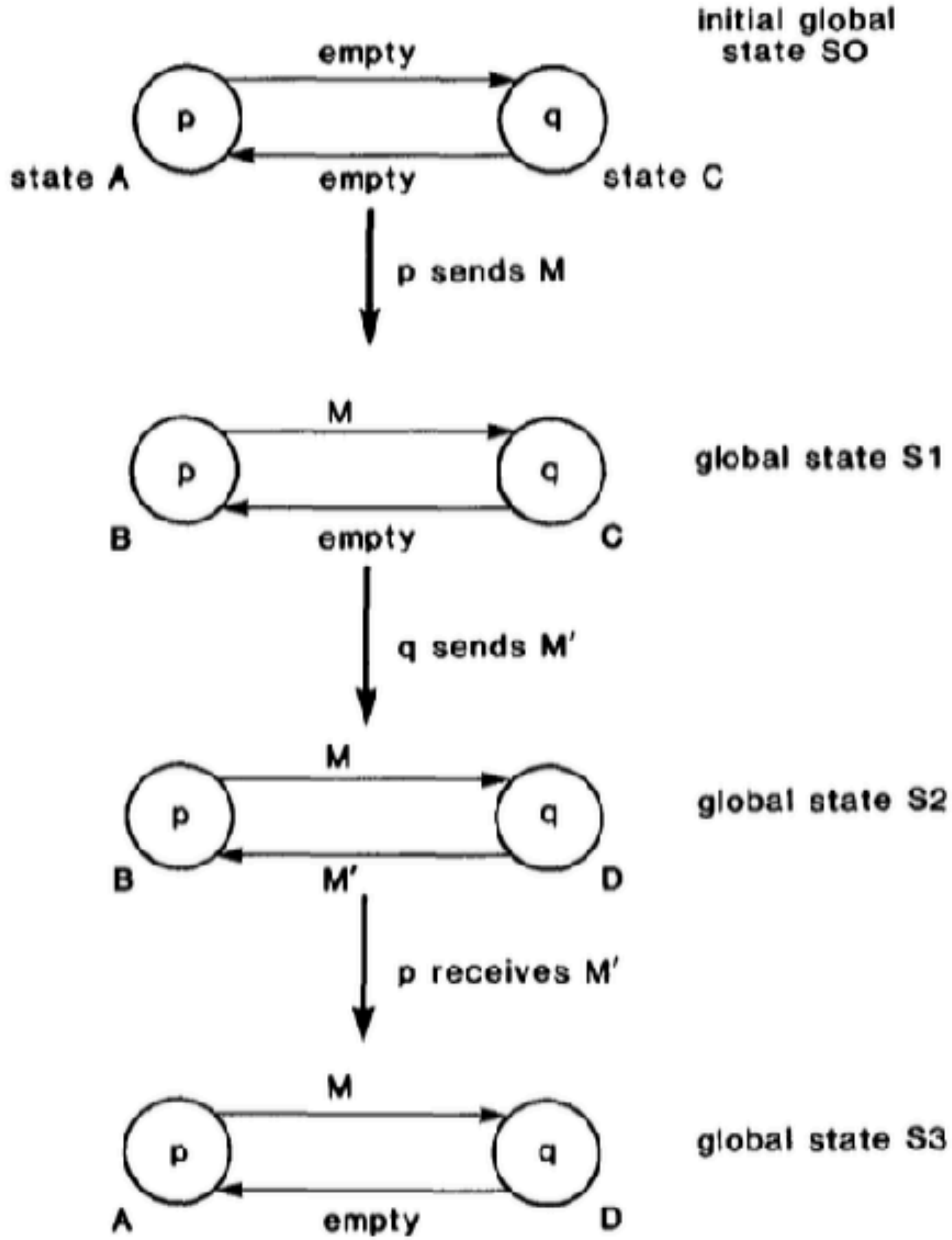


Fig. 7. A computation for Example 2.2.

Chandy Lamport Protocol

- Assumptions:
  - No message remains forever in transit
  - Messages can be delayed but not lost
  - If the graph is not strongly connected, at least one node in each component starts the process

Chandy Lamport Protocol

- Process p0 starts the protocol by sending itself a "take snapshot" message.

- Let pf be the process from which pi receives the "take snapshot" message for the first time. Upon receiving this message, pi records its local state i and relays the "take snapshot" message along all of its outgoing channels. No intervening events on behalf of the underlying computation are executed between these steps. Channel state (f,i) is set to empty and pi starts recording messages received over each of its other incoming channels.

- Let ps be the process from which pi receives the "take snapshot" message beyond the first time. Process pi stops recording messages along the channel from ps and declares channel state si as those messages that have been recorded.

Validating Predicates

- Stable predicates can be faithfully validated
- Unstable predicates are tricky:
  - Algorithm may detect state that never held in an actual run of the distributed computation
  - Predicate may have changed by the time the observer gets to know

Defining predicates

- Possibly(P): There exists a consistent observation O of the computation such that P holds in a global state of O.

- Definitely(P): For every consistent observations O of the computation, there exists a global state of O in which P holds.

Predicates

- Possibly(P) and Definitely (not P) can hold at the same time!
- How? By being true in different global states of the same run

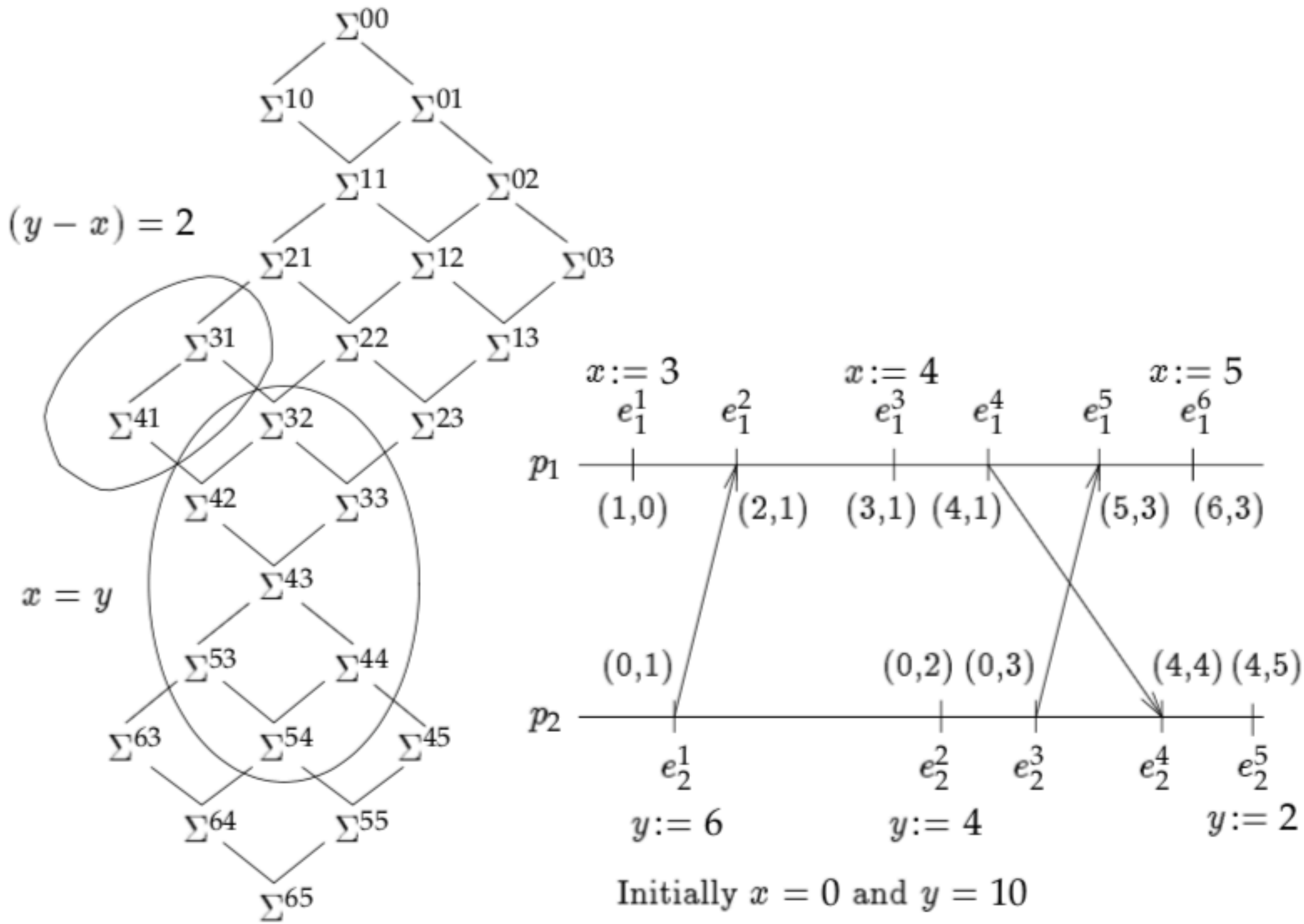


Figure 16. Global States Satisfying Predicates $(x = y)$ and $(y - x) = 2$