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)

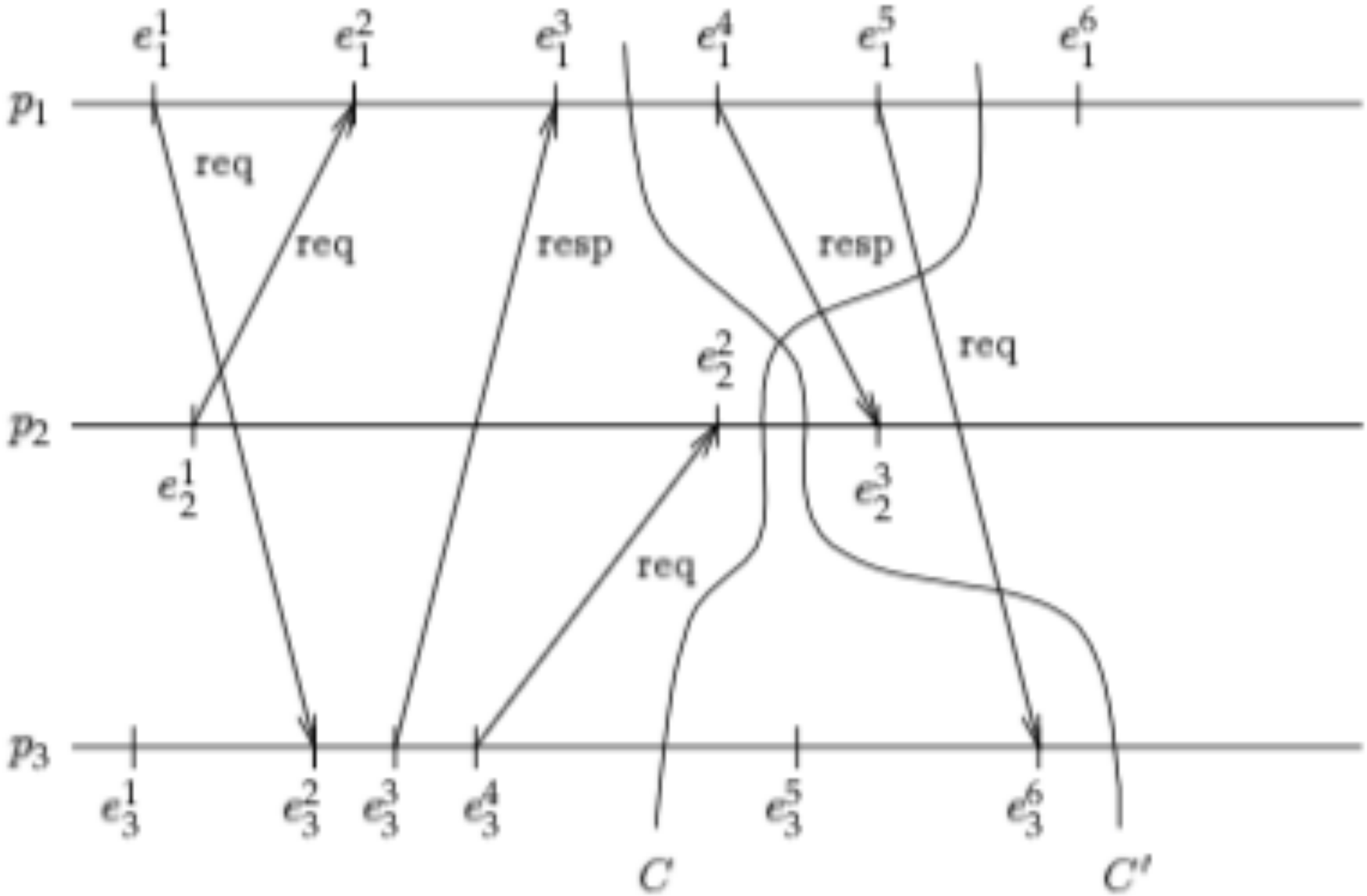


Figure 2. Cuts of a Distributed Computation

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

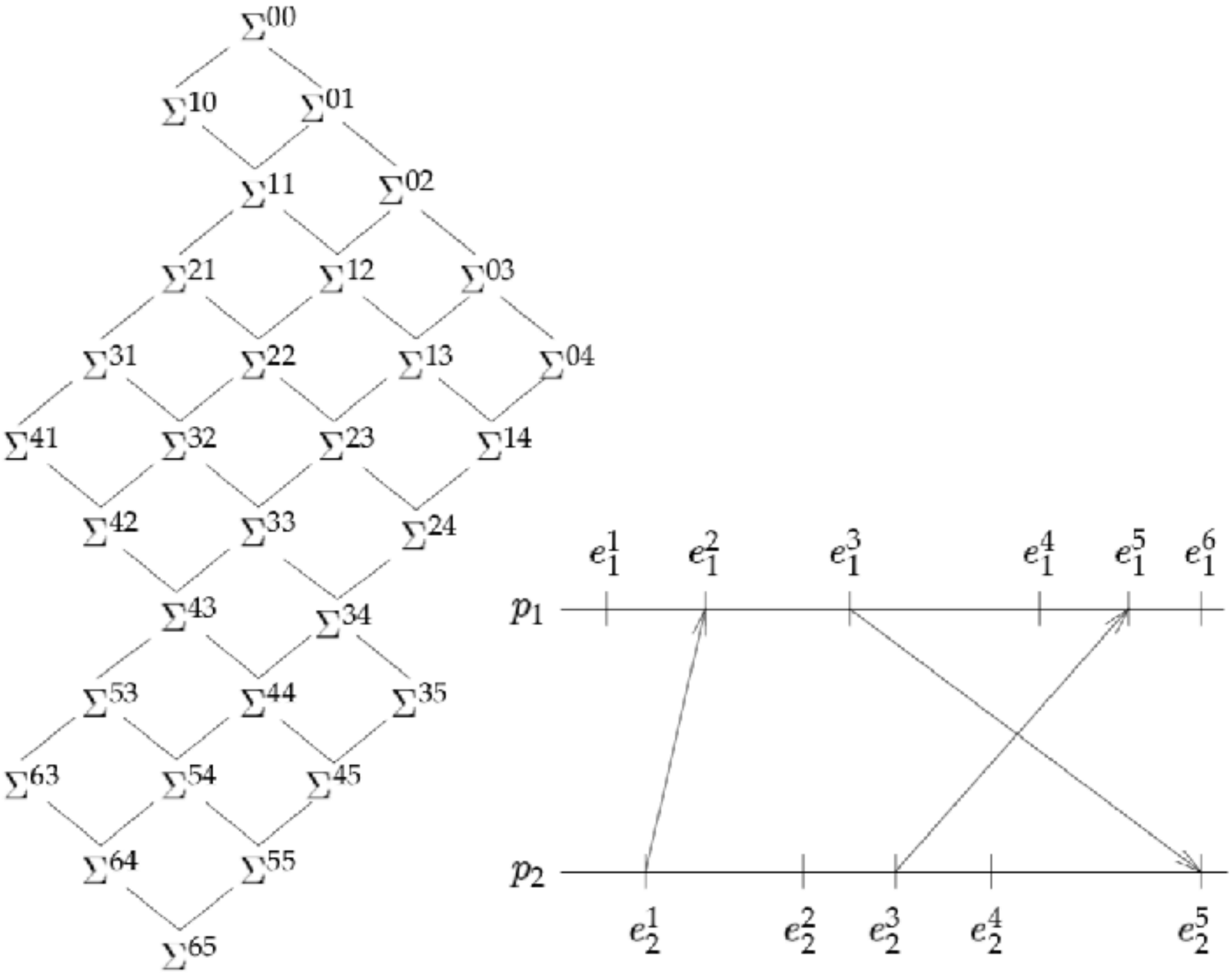


Figure 3. A Distributed Computation and the Lattice of its Global States

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

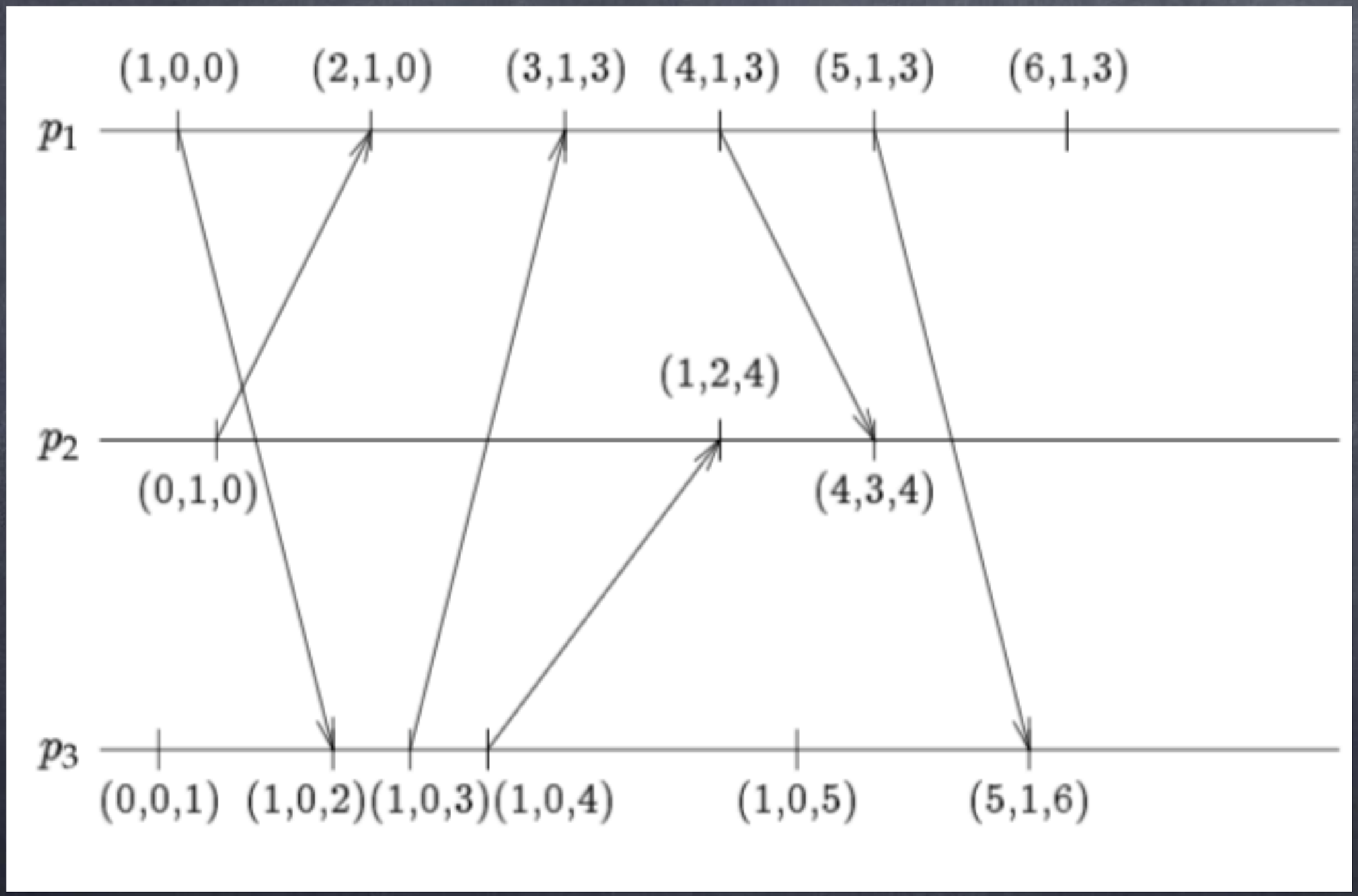
- Need stronger condition to implement Network Controller
- Strong Clock Condition:
  - E1 < E2 => TS(E1) < TS(E2)
  - TS(E1) < TS(E2) => E1 < E2

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)
p_1
(1,2,4)
p_2
(0,1,0)
(4,3,4)
p_3
(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 smaller 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]

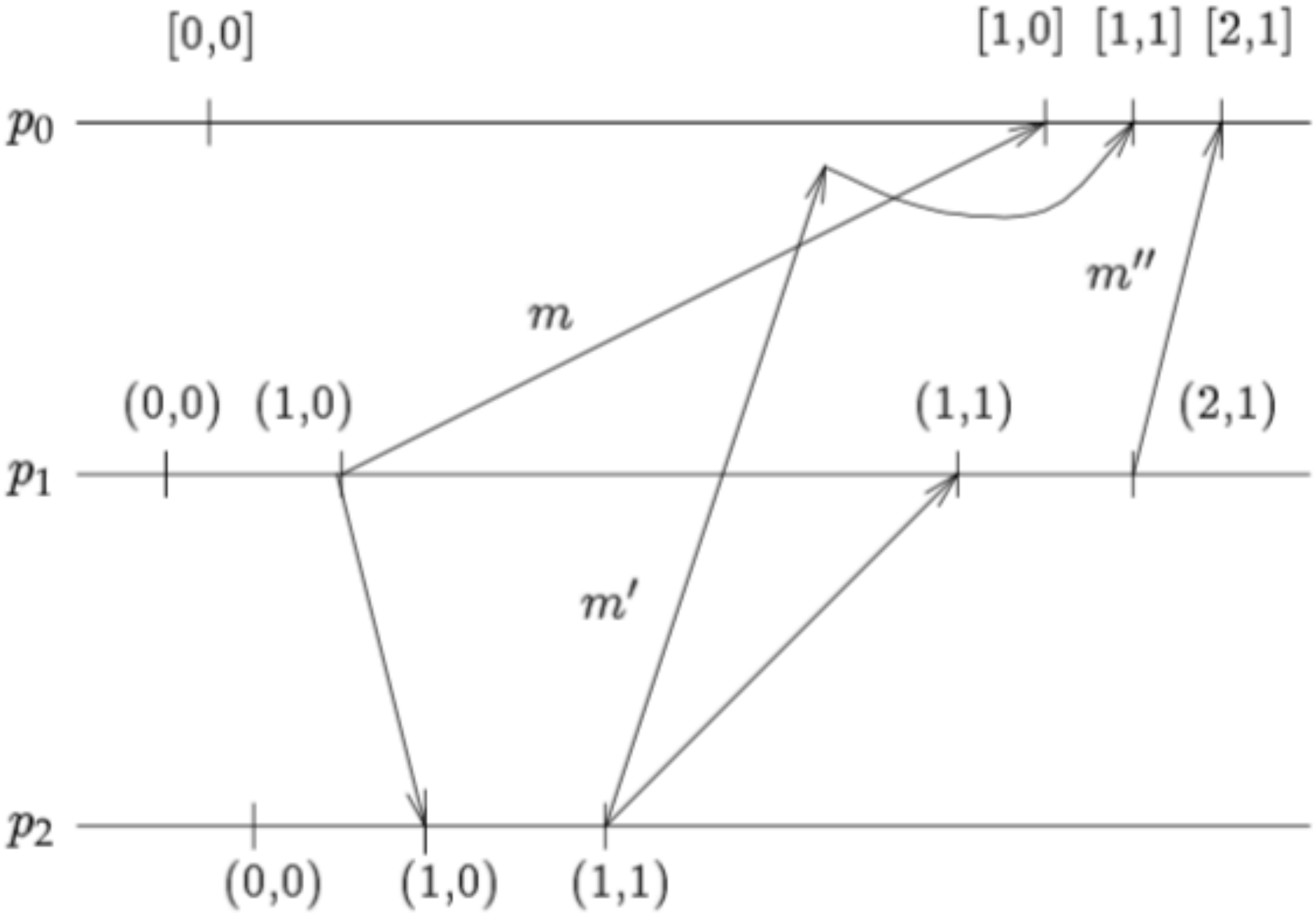


Figure 8. Causal Delivery Using Vector Clocks

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]

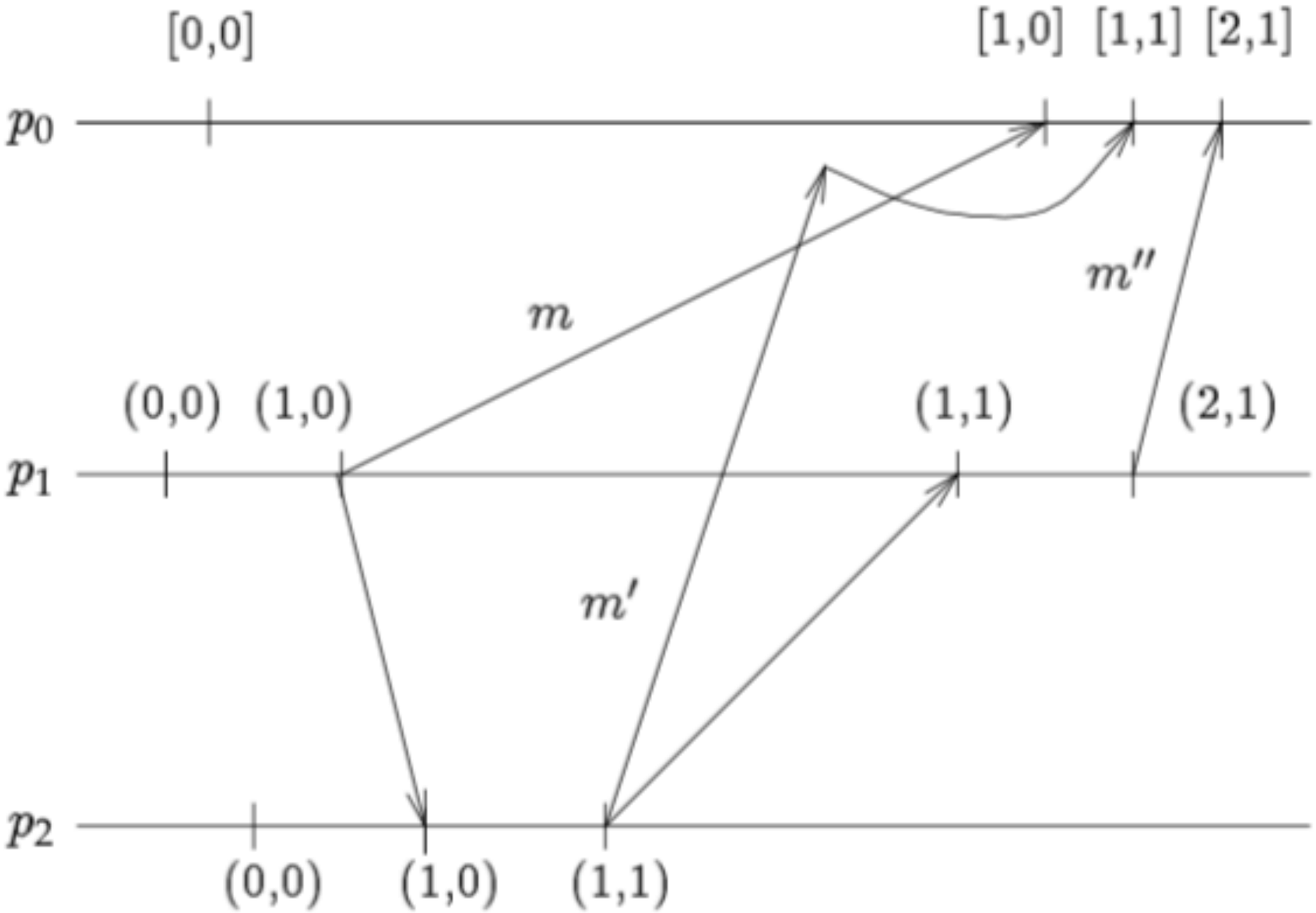


Figure 8. Causal Delivery Using Vector Clocks

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?

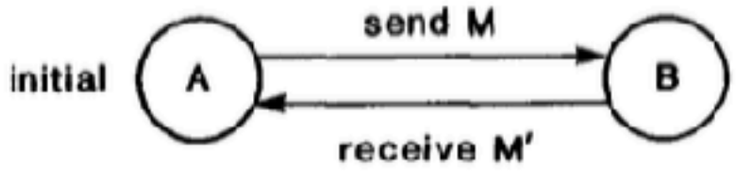


Fig. 5. State-transition diagram for process p in Example 2.2.

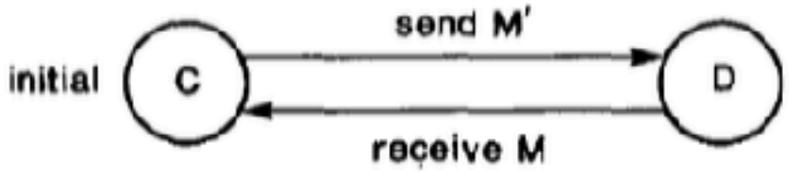


Fig. 6. State-transition diagram for process q in Example 2.2.

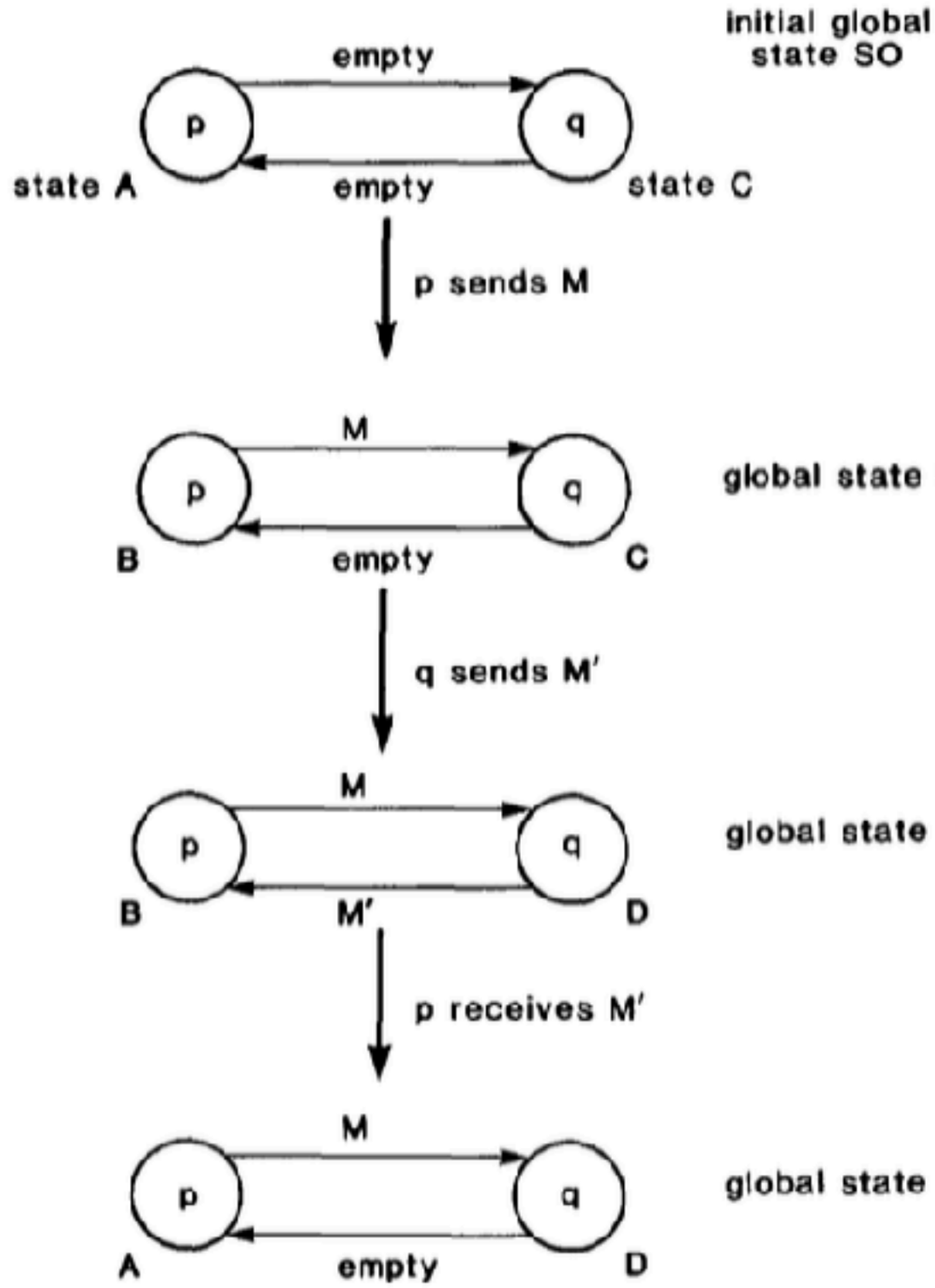


Fig. 7. A computation for Example 2.2.

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

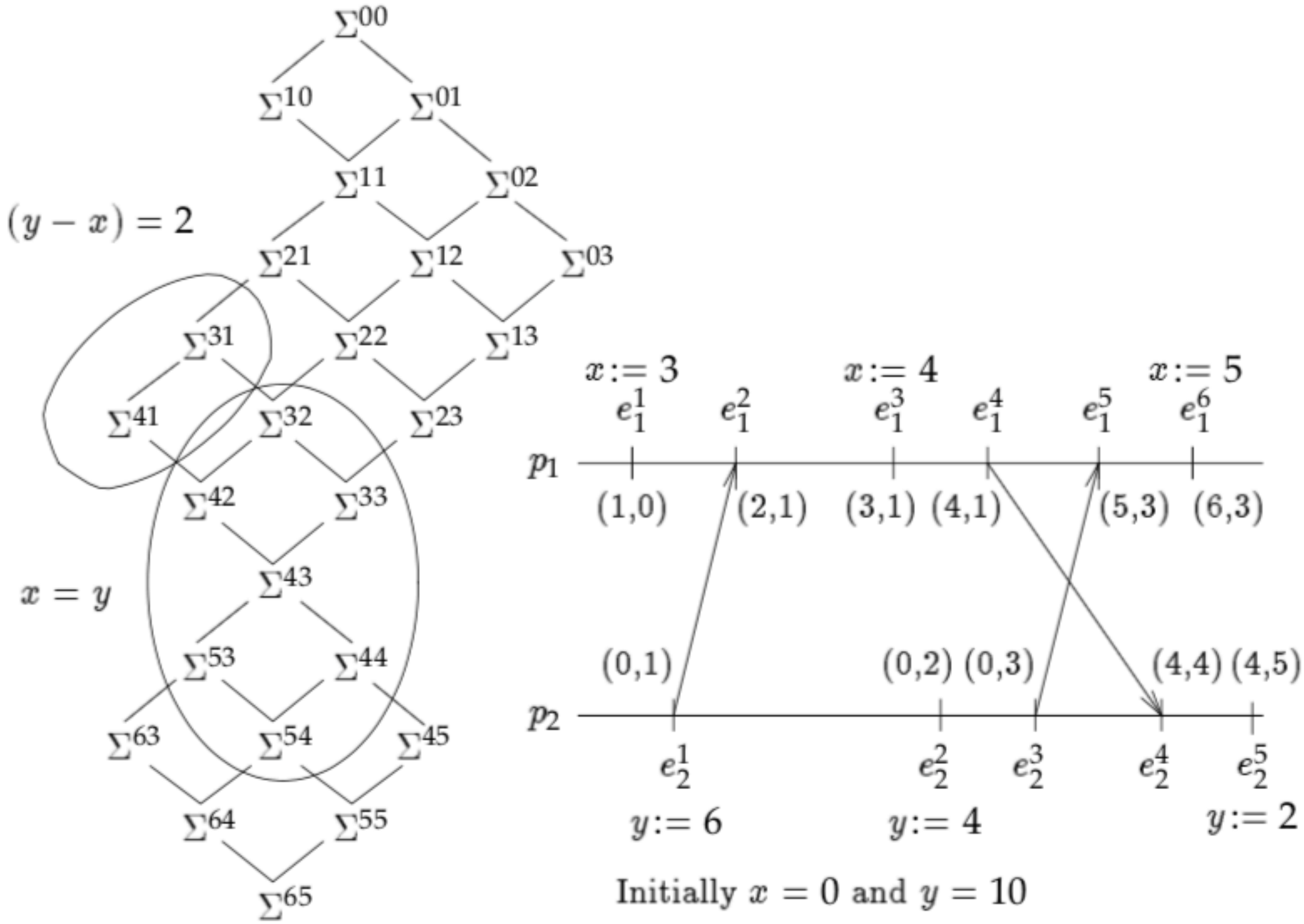


Figure 16. Global States Satisfying Predicates $(x=y)$ and $(y-x)=2$

Feb 11
Safety and Liveness

Vijay Chidambaram

Safety and Liveness

- Safety: "bad things don't happen (ever)"
- Liveness: "good things happen eventually"
- We will formalize these properties to help reason about them
- We will build on the theory we have learnt so far

Property

- Property: A set of infinite sequences of program states
- A program satisfies property P if each of its histories H is a subset of P
- We will use Buchi automata to specify properties
- A buchi automaton M accepts the sequence of program states in the property it specifies

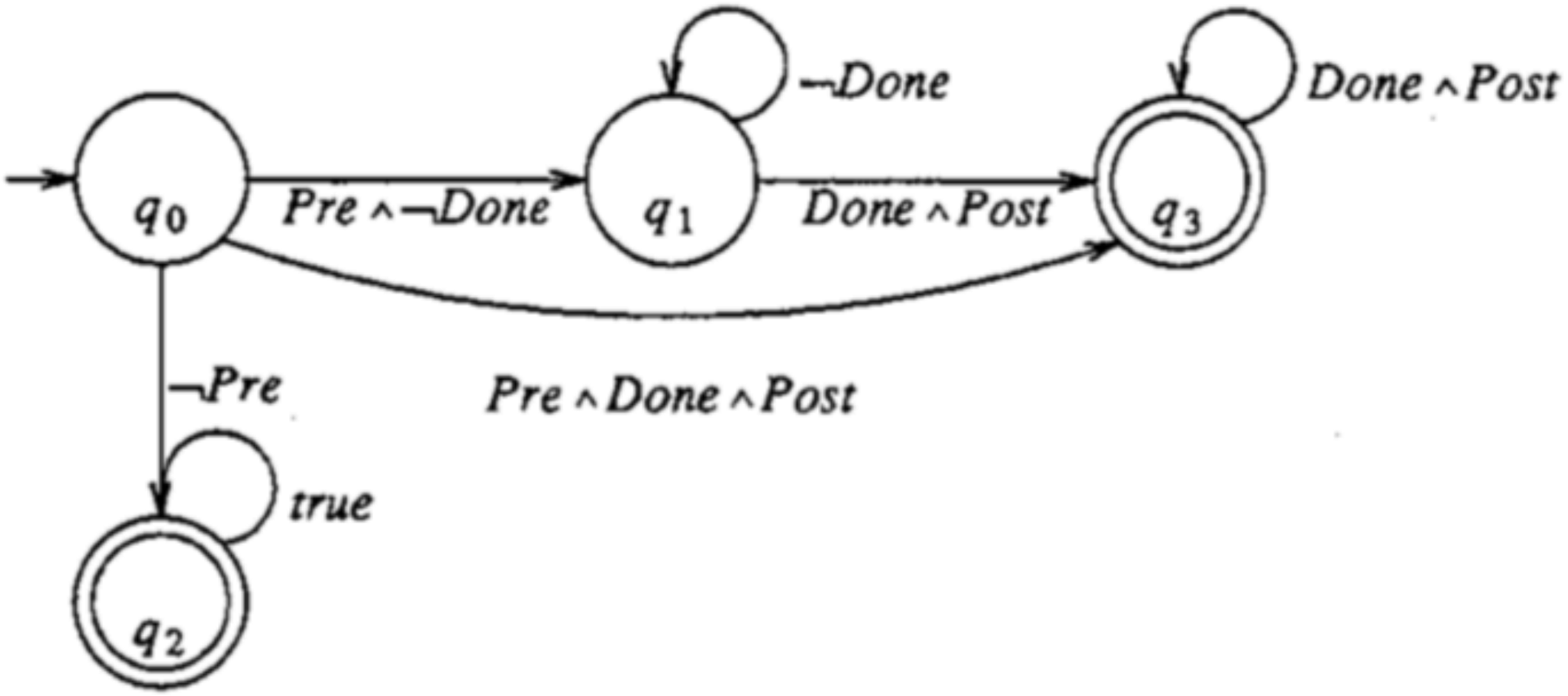

¬Done
Done ∧ Post
q0
Pre ∧ ¬Done
q1
Done ∧ Post
q3
¬Pre
Pre ∧ Done ∧ Post
true
q2

Fig. 1. m_{tc}

Buchi Automatons

- A buchi automaton contains a start state and a set of accepting states
- Arcs between states are labelled with predicates called transition predicates
- If a state does not have an arc for a given program state, we say an undefined transition occurs
- A buchi automaton is reduced if there is a path from every state to an accepting state
- Non-deterministic automatons have multiple start states or multiple transition arcs for the same input

Buchi Automatons

- Formally, a Buchi automaton m for a property of a program rc is a five-tuple (S, Q, Q-start, Q-accept, D)

- S is the set of program states of m,
  Q is the set of automaton states of m,
  D (Q, S) is the transition function of m.

-

Specifying Safety

Safety:

$$(\forall \sigma : \sigma \in S^{\omega} : \sigma \models P$$
$$\Leftrightarrow (\forall i : 0 \leq i : (\exists \beta : \beta \in S^{\omega} : \sigma[..i]\ \beta \models P))), \qquad (3.1)$$

- All runs are a subset of property P
- For a reduced Buchi automaton m, define its closure cl(m) to be the corresponding Buchi automaton in which every state has been made into an accepting state.
- The closure of m can be used to determine whether the property specified by m is a safety property.
- It rejects only by attempting an undefined transition (a "bad thing").
- If m and cl(m) accept the same language then m recognizes a safety property.

Specifying Liveness

- The thing to observe about a liveness property is that no partical execution is irremediable since if some partial execution were irremediable, then it would be a "bad thing" (and thus a safety property)

- A buchi automaton m specifies a liveness property if and only if its closure accepts every input

- For all finite inputs, there exists an infinite sequence of states that result in the property P being maintained

Liveness: $(\forall \alpha : \alpha \in S^* : (\exists \beta : \beta \in S^\omega : \alpha\beta \models P))$. (3.4)

Partioning into safety and liveness

- Given a Buchi automaton m, it is not difficult to construct Buchi automata Safe(m) and Live(m) such that Safe(m) specifies a safety property, Live(m) specifies a liveness property, and the prop- erty specified by m is the intersection of those speci- fied by Safe(m) and Live(m).

- Safe(m) = closure of m

- Live(m) = m augmented with a trap accepting state and all other state transitioning on undefined input to the trap state

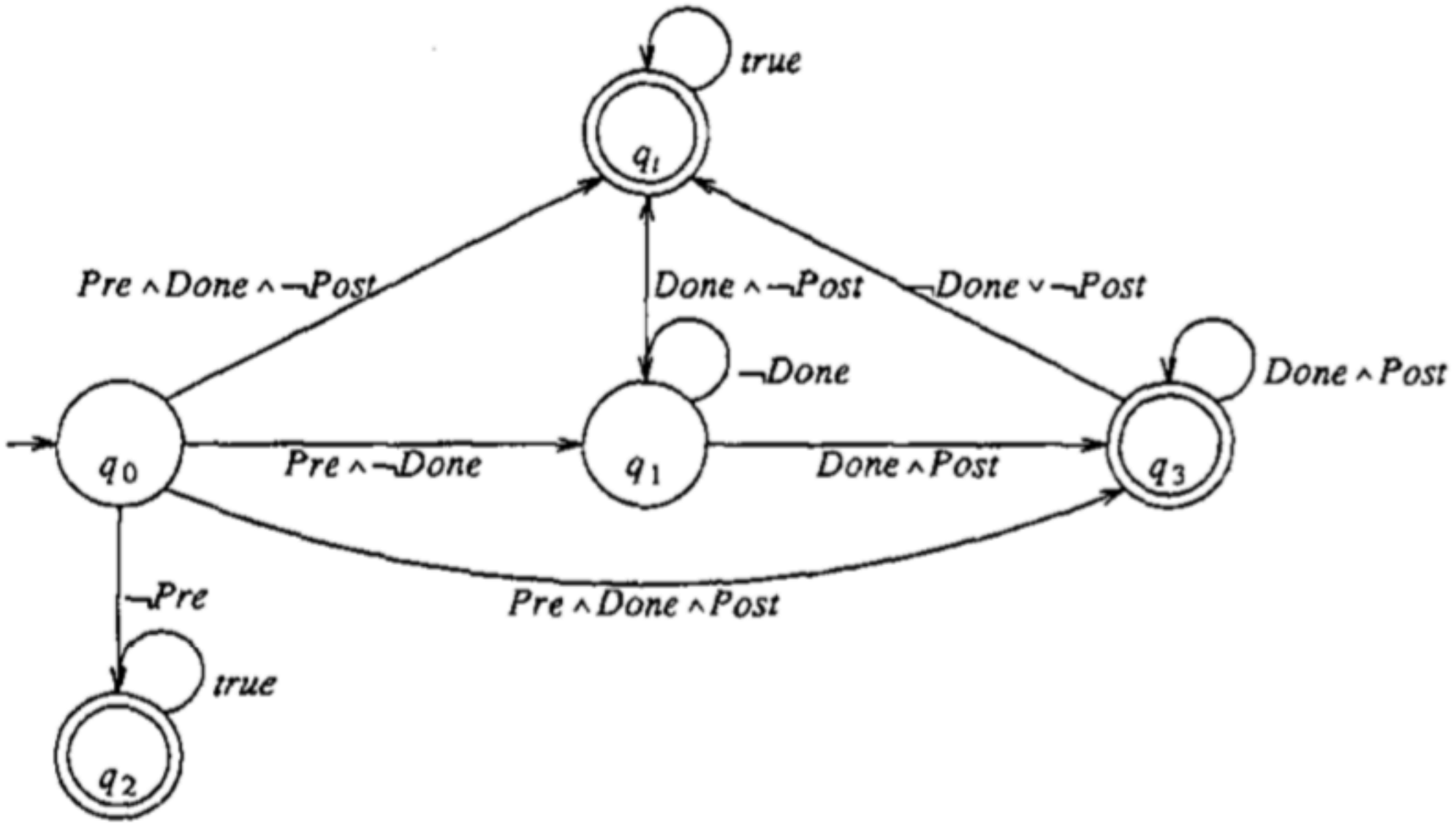


Fig. 8. *Live*(m_{tc})

Feb 13
Distributed Commit Protocols

Vijay Chidambaram

The problem

- We have multiple sites
- Data is divided among these sites (not replicated)
- We want to run a distributed transaction over this distributed data with ACID guarantees:
  - Atomic: all sites are updated with tx results, or none are
  - Consistent: updates are applied in a consistent fashion at all sites
  - Isolation: no tx sees partial results of other concurrent tx
  - Durability: after the tx commits, even if all sites lose power, the data is still available after reboot

Distributed Txs

- Each site has a Transaction Manager (TM)
- Txs are submitted to the TM at their local site
- Read(x) or write(x) forwarded to TM where X lives
- A Tx is Committed or Aborted — decision needs to be taken by all sites
- Incorrect if one site decides to Commit, and another site decides to Abort
- What to do if a tx site fails?

Failures in a distributed system

- A site could fail, or a link connecting sites could fail
- We assume there is a path from every site to every other site
- Failures are fail-stop
- Partial failure: some nodes are up, others are down
  - Partial failures are tricky because sites are unsure of the status of other sites (no common knowledge)
- Total failure: all nodes are down

Network Partition

- Dividing the network into two disconnected graphs, with no messages flowing from one side of the graph to the other
- Can happen due to router/link failures
- Usually temporary
- Once connection is restored, nodes can then talk to each other across the partition

Detecting Failures

- Done via time-out T
- If a node hasn't responded in time T, it is assumed to be dead
- Setting the value of T is tricky
  - If T is too high, we detect failures very late
  - If T is too low, we have false positives where we detect failures spuriously

Conditions to commit

- A Tx T can commit at Site S if:
  - T has read only committed values
  - All the values written by T are durably stored
- Each site contains a distributed coordination log where information is recorded about txs

Atomic Commitment Protocol

- ACI: All processes that reach a decision reach the same one.
- AC2: A process cannot reverse its decision after it has reached one.
- AC3: The Commit decision can only be reached if all processes voted Yes.
- AC4: If there are no failures and all processes voted Yes, then the decision will be to Commit.
- AC.5: Consider any execution containing only failures that the algorithm is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision.

Implications of the conditions

- A process can unilaterally abort the tx by voting No
- Once a process can voted Yes, it cannot later unilaterally decide No
- Uncertainty period: where a process has voted but is uncertain about the outcome (happens only when voting Yes)
- Proposition 7.1: If communication failures or total failures are possible, then every ACP may cause processes to become blocked.
- Proposition 7.2: No ACP can guarantee independent recovery of failed processes.

Two Phase Commit Protocol

- The coordinator sends a VOTE-REQ (i.e., vote request) message to all participants.
- When a participant receives a VOTE-REQ, it responds by sending to the coordinator a message containing that participant's vote: YES or NO. If the participant votes No, it decides Abort and stops.
- The coordinator collects the vote messages from all participants, If all of them were YES and the coordinator's vote is also Yes, then the coordinator decides Commit and sends COMMIT messages to all participants.
- Otherwise the coordinator decides Abort and sends ABORT messages to all participants.
- Each participant that voted Yes waits for a COMMIT or ABORT message from the coordinator. When it receives the message, it decides accordingly and stops.

Augmenting 2PC

- 2PC as outlined before satisfies AC 1-4
- Does not satisfy AC5: ".. if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision."
- To handle failures, each site should durably record its vote and decision in the distributed coordination log
- Termination protocol: uncertain processes contact coordinator to learn of decision
- Cooperative Termination Protocol: uncertain processes contact each other instead of the coordinator

Feb 18
Failure Detectors

Vijay Chidambaram

Recap: models

- Async model: any message or event can take arbitrary time, there are no physical clocks
- Sync model: Messages get delivered within a bounded time, physical clocks can be used
- Async models are more portable, and in practice, timing assumptions in sync models are wrong at least temporarily

Consensus and Atomic Broadcast

- These are two useful primitives
- Consensus: N nodes agree on value X, in the presence of failures
- Atomic Broadcast: N nodes receive the same items in the same order, in the presence of failures
- Consensus and Atomic Broadcast are equivalent problems
- Both impossible to solve under pure asynchrony
  - We will see this result later in class

Failure Detectors

- Consensus impossible in async model because we cannot differentiate a failed node and a slow node
- Propose to solve consensus in asynchronous model by introducing unreliable failure detectors
- Model: async + crash failures + unreliable failure detectors
  - Each process p in the system maintains a list of processes suspected to be crashed

Failure Detectors

- Defined in terms of abstract properties
- Any implementation that provides those properties is then okay
  - Good engineering practice!
- Completeness: every failure is eventually detected
- Accuracy: limits false alarms
- Reducibility: D is reducible to D' if a dist algo can transform D to D'.

W - weakest failure detectors

- Completeness: There is a time after which every process that crashes is permanently suspected by some correct process.
- Accuracy. There is a time after which some correct process is never suspected by any correct process.
- Allows a process to be wrongly identified as crashed by all processes repeatedly
- Allows a process to be added and removed again and again by other processes in their crash list

W failure detector

- Why is this useful? Guarantees safety
- A consensus system built using W gets safety, but not liveness
- No wrong values are accepted, processes don't accept different values

Failure Detector

- Every process q periodically sends a "q-is-alive" message to all
- If a process p times-out on some process q, it adds q to its list of suspects.
- If p laters gets message from q:
  - It removes q from list
  - It increases time-out value
- This does not satisfy W in async system with unbounded timeouts since correct processes be wrongly suspected forever
- However, in a sync system, this does satisfy W

Weakest failure detector - W0

- W0 satisfies the properties of W, and no other properties
- W0 is necessary and sufficient for solving consensus in async systems

The Model

- Async model, reliable communication channels
- A process fails by stopping
- Once a process fails, it does not recover
- crashed() and correct() are the set of failed and active processes
- We assume that not all processes fail at the same time
- H(p, t) = the list of processes p thinks has failed at time t

Failure Detector (FD) Properties

- Strong Completeness: every crashed process is suspected by every correct process, eventually.
- Weak completeness: every crashed process is suspected by some correct process, eventually.
- Strong Accuracy: No process is suspected before it crashes
- Weak Accuracy: Some correct process is never suspected
- If we want strong/weak accuracy to only hold at some time points, we use eventual strong/weak accuracy: after some time point t, strong accuracy holds

Eventual Strong Accuracy

- Eventual Strong Accuracy. There is a time after which correct processes are not suspected by any correct process.
- Eventual Weak Accuracy. There is a time after which some correct process is never suspected by any correct process.

FD Classes

- Perfect (P): strong completeness + strong accuracy
-

Completeness	Accuracy			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually Perfect* $\diamond\mathcal{P}$	*Eventually Strong* $\diamond\mathcal{S}$
Weak	$\mathcal{Q}$	*Weak* $\mathcal{W}$	$\diamond\mathcal{Q}$	*Eventually Weak* $\diamond\mathcal{W}$

FIG. 1. Eight classes of failure detectors defined in terms of accuracy and completeness.

- We can transform a failure detector with weak completeness into one that satisfies strong completeness

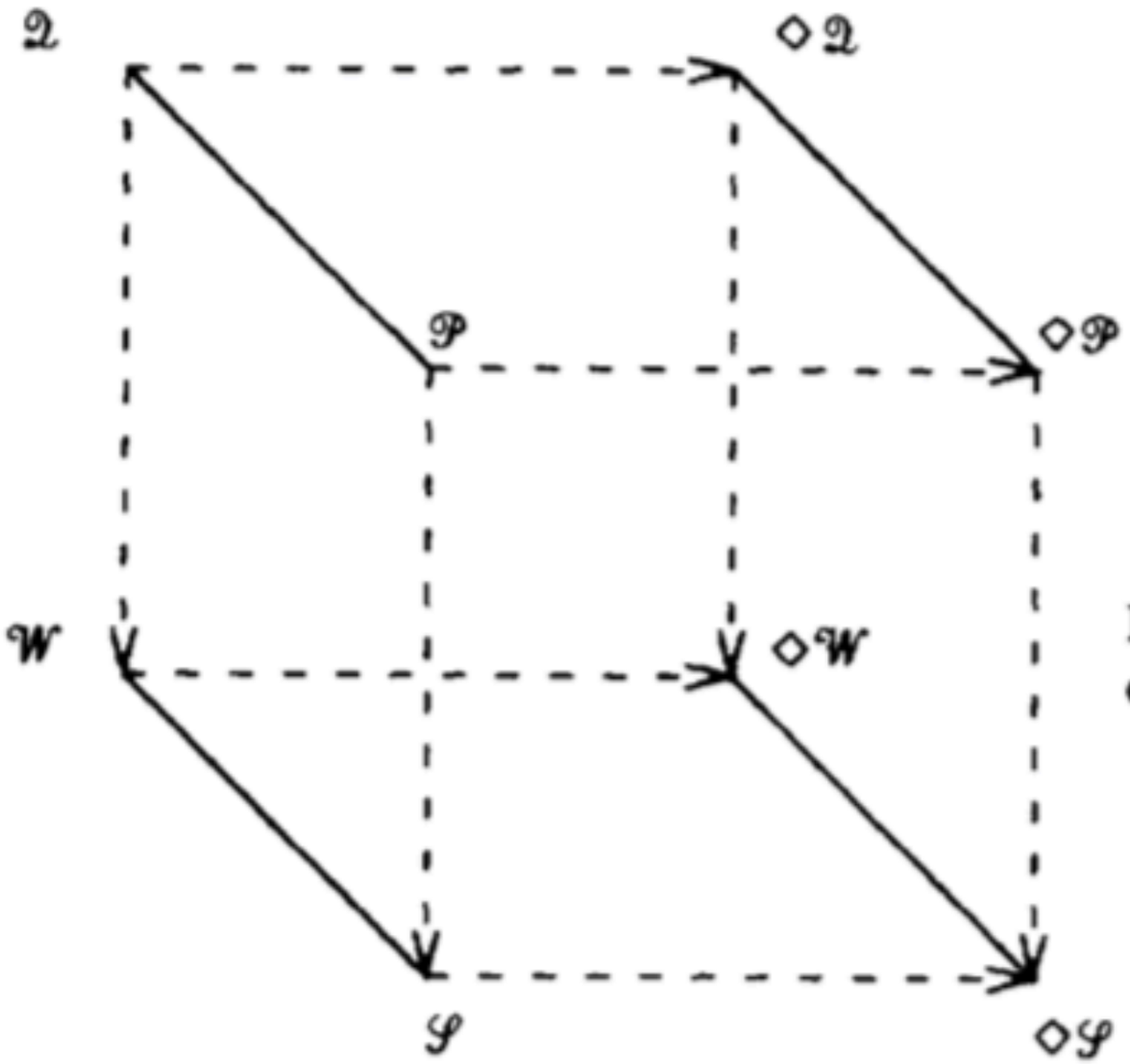


FIG. 8. Comparing the eight failure detector classes by reducibility.

C - - - -> C': C' is strictly weaker than C'

C ——— C': C is equivalent to C'

Transforming WC to SC

- WC: Some process suspects every crashed process
- SC: Every process suspects every crashed process
- Transformation algo:
  - Each process p broadcasts suspicion of crashed process q
  - On receiving a message from p about q:
    - Add q to suspects list
    - Remove p from suspects list

FD Classes

- Now we have only four classes, all with Strong C:
  - Perfect: Strong A
  - Strong: Weak A (poor naming)
  - Eventually Perfect: Eventually Strong A
  - Eventually Strong: Eventually Weak A

Reliable Broadcast

- Guarantees:
  - all correct processes deliver the same set of messages,
  - all messages broadcast by correct processes are delivered,
  - no spurious messages are ever delivered.
- Properties:
  - Validity. If a correct process R-broadcasts a message m, then it eventually R-delivers m.
  - Agreement. If a correct process R-delivers a message m, then all correct processes eventually R-deliver m.
  - Uniform integrity. For any message m, every process R-delivers m at most once, and only if m was previously R-broadcast by sender(m).

Reliable Broadcast

- R-broadcast (m):
  - Send m to all (including sender p)
- R-deliver (m):
  - If getting m for the first time:
    - If sender(m) != p then send m to all
    - Deliver(m)

Consensus

- All correct processes propose a value and must agree on a proposed value
- Termination. Every correct process eventually decides some value.
- Uniform integrity. Every process decides at most once.
- Agreement. No two correct processes decide differently.
- Uniform validity. If a process decides v, then v was proposed by some process.

Solving Consensus using Strong FD

- FD has strong completeness but weak accuracy
  - every crashed process is suspected by every correct process, eventually.
  - Some correct process is never suspected

Every process p executes the following:

procedure *propose*(v_p)
  $V_p \leftarrow (\perp, \perp, \ldots, \perp)$ {*p's estimate of the proposed values*}
  $V_p[p] \leftarrow v_p$
  $\Delta_p \leftarrow V_p$

  Phase 1: {*asynchronous rounds* r_p, $1 \leq r_p \leq n-1$}
    for $r_p \leftarrow 1$ to $n-1$
      send (r_p, Δ_p, p) to all
      wait until $[\forall q:$ received (r_p, Δ_q, q) **or** $q \in \mathcal{D}_p]$ {*query the failure detector*}
      $msgs_p[r_p] \leftarrow \{(r_p, \Delta_q, q) \mid$ received $(r_p, \Delta_q, q)\}$
      $\Delta_p \leftarrow (\perp, \perp, \ldots, \perp)$
      for $k \leftarrow 1$ to n
        if $V_p[k] = \perp$ **and** $\exists (r_p, \Delta_q, q) \in msgs_p[r_p]$ **with** $\Delta_q[k] \neq \perp$ **then**
          $V_p[k] \leftarrow \Delta_q[k]$
          $\Delta_p[k] \leftarrow \Delta_q[k]$

Phase 2: send V_p to all
    wait until $[\forall q:$ received V_q **or** $q \in \mathcal{D}_p]$ {*query the failure detector*}
    $lastmsgs_p \leftarrow \{V_q \mid$ received $V_q\}$
    for $k \leftarrow 1$ to n
      if $\exists V_q \in lastmsgs_p$ **with** $V_q[k] = \perp$ **then** $V_p[k] \leftarrow \perp$

Phase 3: *decide*(first non-$\perp$ component of V_p)

FIG. 5. Solving Consensus using any $\mathcal{D} \in \mathcal{S}$.

Algo

- Phase 1:
  - processes execute n-1 async rounds
  - In each round, p broadcast their proposed values
    - In each around, p waits for all other processes not suspected to be crashed
- At the end of Phase 2, everyone agrees on a vector of proposed values
- Phase 3: decided based on first non-null component of vector

Results

- With Strong Failure Detector, we could solve consensus
  - Key is that one correct process is never suspected of being crashed
- We can also solve consensus with weaker failure detectors
  - A FD with weak completeness and eventual weak accuracy is the weakest FD that can be used
  - Weakest FD can solve consensus as long as less than half the processes are faulty

Why are failure detectors important?

- In previous algo, each process p waits for a message for all non-crashed processs
- The failure detector helps identify this set
- Without FD, process p may block forever waiting for process q
- Strong completeness of FD ensures that every crashed process is suspected eventually -> this eliminates the endless waiting

Feb 20

FALCON and RAPID

Vijay Chidambaram

Detecting Failures

- Detecting failures in distributed systems is important
- Often, detecting failures takes longer than the recovery
- Falcon provides:
  - sub-second detection in the common case
  - No false positives
  - Terminates minimum needed component to ensure results

Time outs in real systems

- Real time-outs:
  - Google File System: 60 seconds
  - Chubby: 12 seconds
  - Dryad: 30 seconds
  - Network File System: 60 seconds

The problem

- Picking time outs correctly is hard
- Achieving a failure detector with three properties simultaneously is hard:
  - Quick detection
  - Reliability
  - Minimal disruption

Reliability

- If the failure detector only cared about reliability, it could kill any process it suspects to be down
  - Therefore making sure in the process
- This is formally called STONITH: Shoot the Other Node in the Head

FALCON

- Fast And Lethal Component Observation Network (FALCON) :)
- Good properties for a Failure Detector:
  - If a node is up, it should be reported as up
  - When a node is down, it should be reported as down after a while
  - The gap to detect down nodes should be low
  - The failure detector should not cause disruption

Core Insight

- Failures are observed quickly if we look at the right software layer
- A process that core dumps will disappear from the process table
- after an operating system panics, it stops scheduling processes
- if a machine loses power, it stops communicating with its attached network switch.
- If the failure detector infiltrates various layers in the system, it can provide reliable failure detection using local instead of end-to-end timeouts and sometimes without using any timeouts.

FALCON

- Spy modules are inserted within various software layers
- If a spy suspects a process is down, it KILLS it
- Thus, we have quick detection and reliability
- Handles crashes of all nodes, but not handle byzantine failures

FALCON

- But spies are embedded within layers, so if a layer goes down, so does a spy
- How do we handle this?
- Answer: we make spies monitor other spies

FALCON Spy Network

- Spies are arranged in a chained network
- Spy monitors the spy in layer above
- OS spy monitors the application spy
- FALCON assumes that spies will miss detecting some failures: backup is a large time-out
- If a network partition happens, FALCON is paused (can't communicate with remote spies)

FALCON Architecture

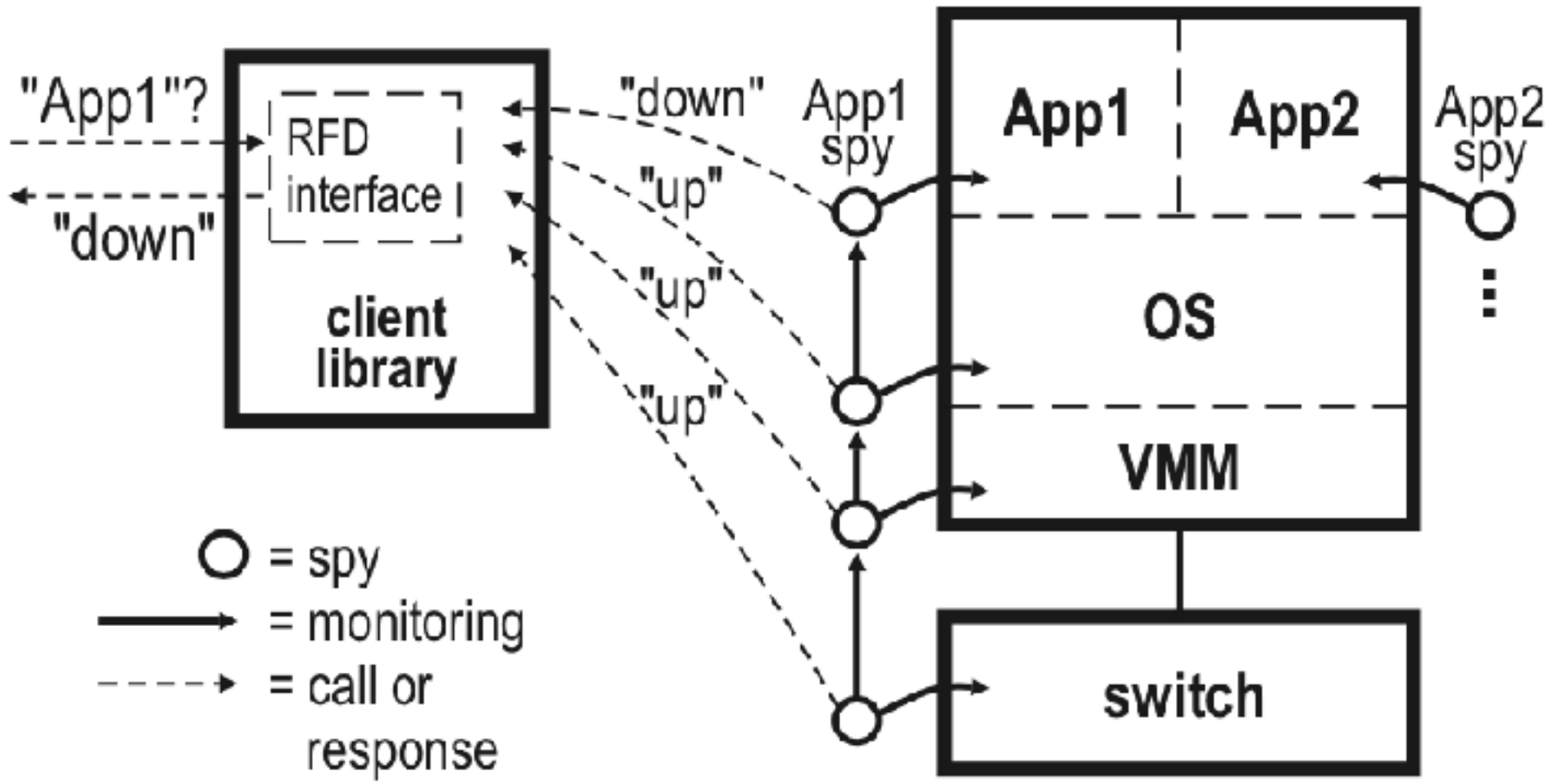


Figure 1—Architecture of Falcon. The application spy provides accurate information about whether the application is up; this spy is the only one that can observe that the application is working. The next spy down provides accurate information not only about its layer but also about whether the application spy is up; more generally, lower-level spies monitor higher-level ones.

FALCON Interface

function	description
init(*target*)	register with spies
uninit()	deregister with spies
query()	query the operational status
set_callback(*callback*)	install callback function
clear_callback()	cancel callback function
start_timeout(*timeout*)	start end-to-end timeout timer
stop_timeout()	stop end-to-end timeout timer

Figure 2—Falcon RFD interface to clients.

FALCON Spy Interface

- Register() - set callback to client
- Cancel() - cancel callback
- Kill() - kill monitored layer

Implementation

- Spies at four layers: OS, Application, VMM, network switch
- Integrated into an application with tens of lines of code

Results

- Added Falcon to ZooKeeper
  - Increases availability by 6x
- Added Falcon to replication library
- Falcon simplifies applications since they can build on top of reliable failure detection

RAPID

- Real-world failures are more than crashes:
  - misconfigured firewalls
  - one-way connectivity loss
  - flip-flops in reachability
  - some-but-not-all packets being dropped
- Existing tools take too long to converge to stable state

The problem

- Detecting membership in a stable manner
- When membership, cluster does heavy-duty operations such as moving data around
- We want to keep this to a minimum

Existing membership schemes

- Logically centralized services
- Gossip based services
  - Can diverge!
- Census: gossip + consistency

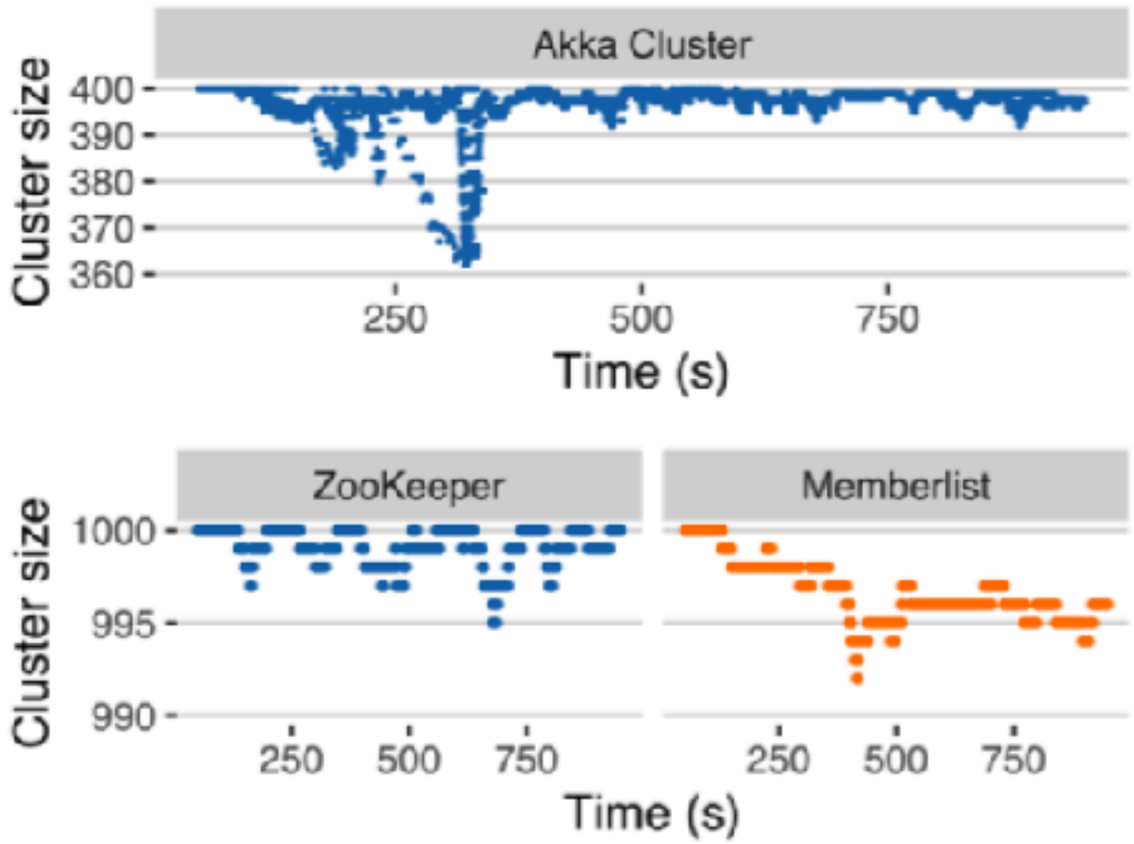


Figure 1: Akka Cluster, ZooKeeper and Memberlist exhibit instabilities and inconsistencies when 1% of processes experience 80% packet loss (similar to scenarios described in [19, 49]). Every process logs its own view of the cluster size every second, shown as one dot along the time (X) axis. Note, the y-axis range does not start at 0. X-axis points (or intervals) with different cluster size values represent inconsistent views among processes at that point (or during the interval).

RAPID Overview

- Have a set of observer processes
- Arrange them as part of an expander overlay graph
- A subject is monitored by multiple observers
- Multiple observers indicating failure is taken as high-fidelity signal
- Observers detect a cut: multiple nodes that must be dropped from membership
- RAPID waits for multiple observers to detect same cut

RAPID

- A configuration in Rapid comprises a configuration identifier and a membership-set (a list of processes).
- Every process p (a subject) is monitored by K observer processes. If L-of- K correct observers cannot communicate with a subject, then the subject is considered observably unresponsive.

Topology

- Rapid organizes processes into a monitoring topology that is an expander graph
- We use the fact that a random K-regular graph is very likely to be a good expander for K ≥ 3
- We construct K pseudo- randomly generated rings with each ring containing the full list of members.
- A pair of processes (o, s) form an observer/subject edge if o precedes s in a ring

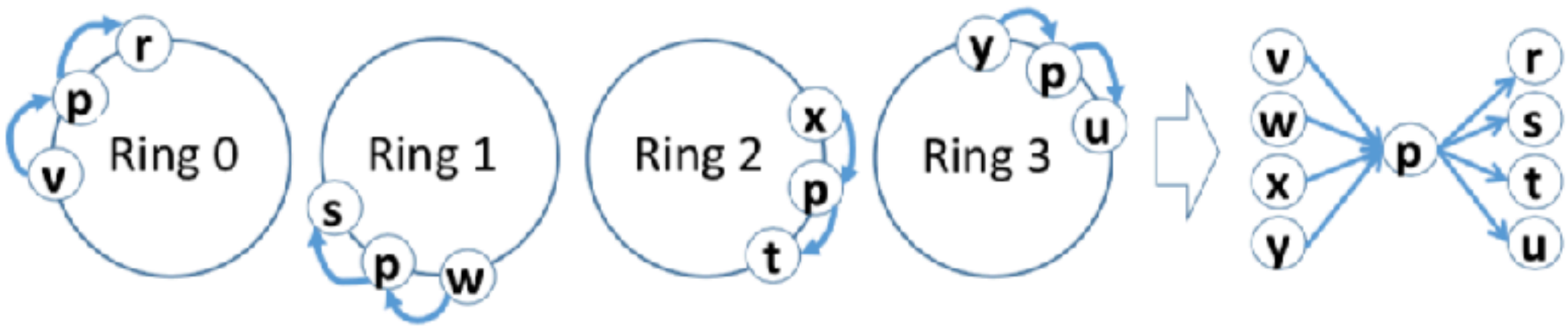


Figure 2: p's neighborhood in a $K = 4$-Ring topology. p's observers are $\{v, w, x, y\}$; p's subjects are $\{r, s, t, u\}$.

Figure 3: Solution overview, showing the sequence of steps at each process for a single configuration change.

Why this topology?

- Strong connectivity: if F out of V processes are fault, there will be (V-F/F) edges from the fault subset to the rest of the processes
- As a result, failures are detected by observers with high probability
- Every process observes K processes, and is observed by K processes. Bounds overhead to O(K)
- Every process joining or leaving results in 2*K edges being added or removed (2 edges each in K graphs)

Multi-process Cut Detection

- Each process gets many alerts
  - Alerts are aggregated until a stable multi-process cut is detected
  - A process defers a decision on a single process until the alert-count it received on all processes is considered stable.
- In particular, it waits until there is no process with an alert-count above a low-watermark threshold L and below a high-watermark threshold H.
- Notifications below L - noise
- Notifications above H - stable decision about process

Proposing Config Changes

- Delay proposing a configuration change un- til there is at least one process in stable report mode and there is no process in unstable report mode

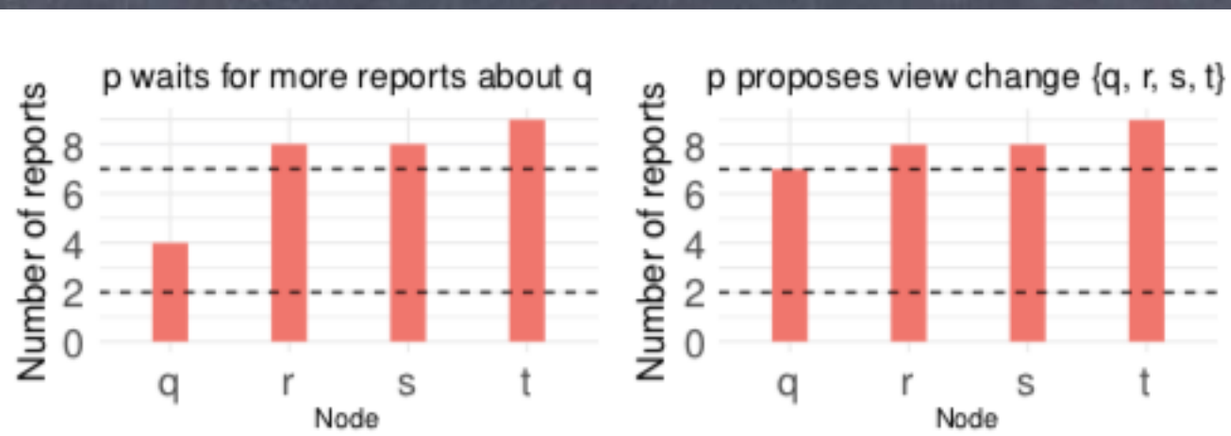


Figure 4: Almost everywhere agreement protocol example at a process p, with tallies about q,r,s,t and $K = 10, H = 7, L = 2$. K is the number of observers per subject. The region between H and L is the unstable region. The region between K and H is the stable region. **Left:** $stable = \{r,s,t\}$; $unstable = \{q\}$. **Right:** q moves from *unstable* to *stable*; p proposes a view change $\{q,r,s,t\}$.

Config Changes

- Once nodes propose a config change, it is applied using a consensus algorithm
- RAPID uses Fast Paxos
- The proposed config change is used as input
- If two thirds of the processes have identical input, Fast Paxos reaches a decision
- Due to the almost-everywhere agreement in RAPID, Fast Paxos reaches decisions quickly

Results

- RAPID brings up a 2000 node cluster 2-5.8x faster than ZooKeeper or Memberlist
- Robust to node crashes, network partitions, etc

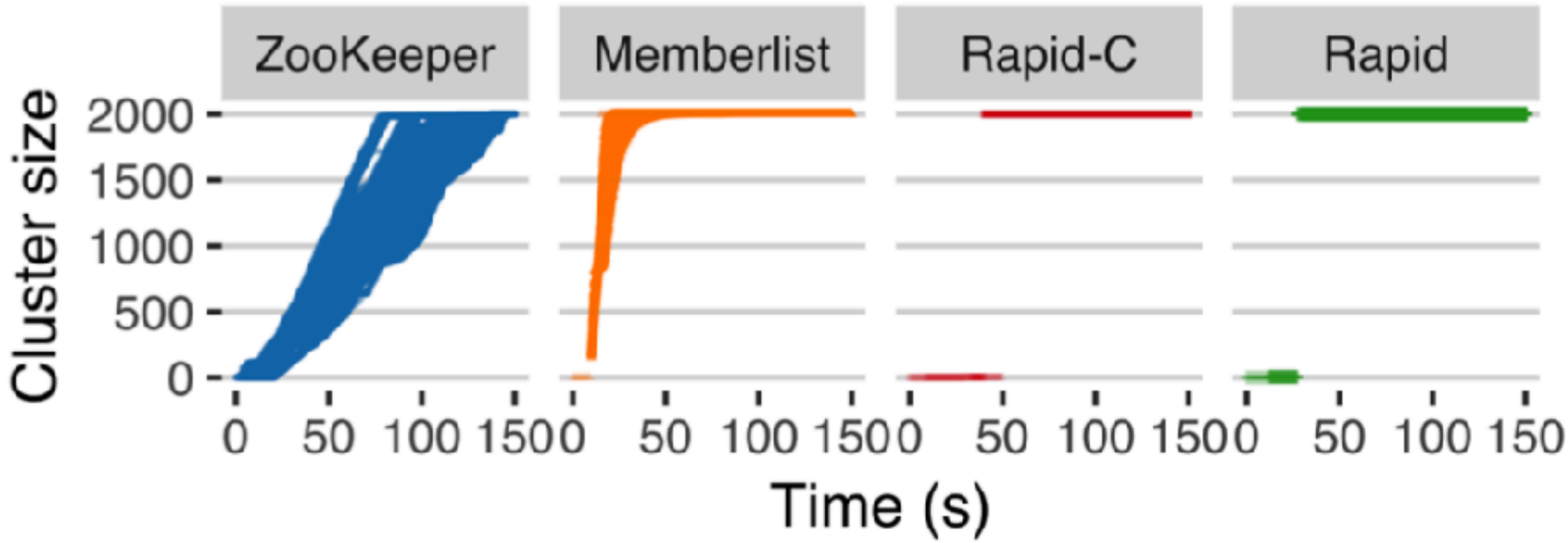


Figure 7: Timeseries showing the first 150 seconds of all three systems bootstrapping a 2000 node cluster.

Feb 25
Chain Replication and Remus

Vijay Chidambaram

The problem

- We want a build a storage system over a group of servers
- Each server may fail at any time
- How do we ensure that the data is consistent when reading and writing to this distributed storage system?
- Pre-cursor to the Amazon S3 we have today

Goals

- Availability:
  - Loss of one server should not render data unavailable
- Consistency:
  - Read of data item X should return the last written value V
  - Write and update operations are serialized in some order
- Performance:
  - High read and write throughput

State is:

$Hist_{objID}$: **update request sequence**

$Pending_{objID}$: **request set**

Transitions are:

T1: Client request r arrives:

$Pending_{objID} := Pending_{objID} \cup \{r\}$

T2: Client request $r \in Pending_{objID}$ ignored:

$Pending_{objID} := Pending_{objID} - \{r\}$

T3: Client request $r \in Pending_{objID}$ processed:

$Pending_{objID} := Pending_{objID} - \{r\}$

if $r = \mathsf{query}(objId, opts)$ **then**

reply according options $opts$ based on $Hist_{objID}$

else if $r = \mathsf{update}(objId, newVal, opts)$ **then**

$Hist_{objID} := Hist_{objID} \cdot r$

reply according options $opts$ based on $Hist_{objID}$

Figure 1: Client's View of an Object.

Correctness

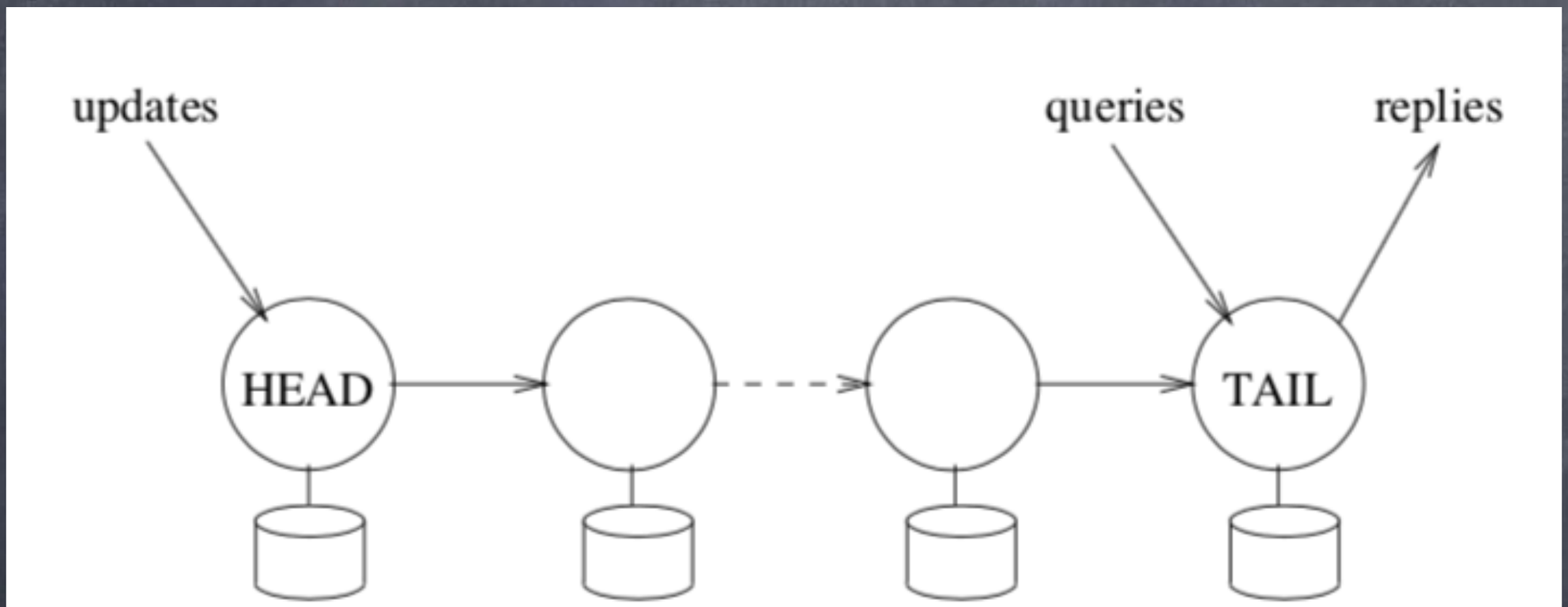
- Correctness depends on maintaining Pending and History

- Pending = set of requests received by any server in the chain and not yet processed by the tail

- History = set of updates done on an object by the chain (by the tail)

Assumptions

- Servers fail in a fail-stop manner
- Server failures can be detected
- Servers communicate via reliable, FIFO channels
- At most n-1 servers (out of n servers) fail concurrently at any time


updates
queries
replies
HEAD
TAIL

Chain Replication

- Properties:
  - Availability: data is available as long as at least one server is up
  - Strong consistency: all read and write requests handled serially by the tail server

Chain Replication

- Servers are arranged in a chain
- Reply for every request is generated by the tail
- Read requests redirected directly to the tail
- Write/update requests arrive at the head, and are atomically processed by each node in the chain

State Forwarding

- The write/update operation is performed first at the head
- The head server performs the operation and produces the resulting low-level update to data
- Every other server in the chain only applies this low-level update; it does not perform the write/update operation
- Thus, if the update operation is non-deterministic, the head operation performs it and changes it into a deterministic low-level update

Dealing with server failures

- A master server:
  - detects failures
  - informs nodes in chain about failed node
  - informs clients of new head or tail if required
- Master implemented using Paxos on a group of nodes

Correctness

Update Propagation Invariant. For servers labeled i and j such that $i \leq j$ holds (i.e., i is a predecessor of j in the chain) then:

$$Hist^{j}_{objID} \preceq Hist^{i}_{objID}.$$

- This invariant is maintained because i is updated before j in the chain. Thus, the history of i = history of j + some updates

Failure of the head

- Clients need to be told of new head
- Pending needs to be updated, whatever was pending at previous head is lost
- Client sees this as a message lost/ignored by the chain

Failure of the tail

- Clients need to be told of new tail
- History needs to be updated, any updates performed by the new tail are considered done by the chain
- Pending needs to be updated, to remove items that have been processed by the new tail
- This works because of the Update Propagation Invariant: new tail is guaranteed to have processed anything processed by old tail

Failure of other servers

- If Server F fails:
  - Master informs F's successor S+
  - Masters informs F's predecessor S-
  - S- forwards to S+ requests that were lost when F failed
- Each server maintains Sent variable
  - Sent = Sent U r when r is forwarded
  - Sent = Sent - r when r's ack is received
  - Requests are forwarded head to tail
  - Acks are forwarded tail to head

Adding new servers

- New servers are added to the tail
- Old tail forwards History to new tail
  - This can be done incrementally since we only need to ensure new tail history is a prefix of old tail history
- Old tail is informed its not the tail
- Old tail forwards requests to new tail
- Master is notified that new tail is functioning
- Clients are notified of new tail

Primary/backup

- Chain replication is an instance of the primary/backup approach
- In primary/backup approach:
  - the primary sequences all requests
  - primary has to wait for acks from all other nodes before responding to client
- In chain replication, sequencing is split between head and tail
  - Head sequences updates
  - Tail sequences queries
- In chain replication, queries are never blocked by updates
  - Can always happen at the tail
- Chain replication has higher latency for writes (must wait until chain processes write in a serial fashion)
  - Primary/backup approach can write in parallel to replicas

Unavailability during failure

- Failure of head/tail:
  - Processing queries blocked for 2 messages while new head is being setup
  - Message 1: master broadcasts new head info to everyone
  - Message 2: master tells all clients about new head
- Middle server failure:
  - Query processing not interrupted
  - Update processing delayed while chain is reconfigured
  - Updates not lost as at least head has update
- Primary/backup failure:
  - Primary failure: 5 message delay
  - Backup failure: 1 message delay

Remus: the problem

- Bring high availability to the masses!
- Goals:
  - Generality and transparency (should work with unmodified apps and unmodified operating systems)
  - Shouldn't require custom hardware
  - No externally visible state should ever be lost
  - Quick failure recovery: should seem like packet loss
  - Crash should leave data in a consistent state

Approach

- VM-Based Whole-system replication
- Speculative Execution
- Async replication

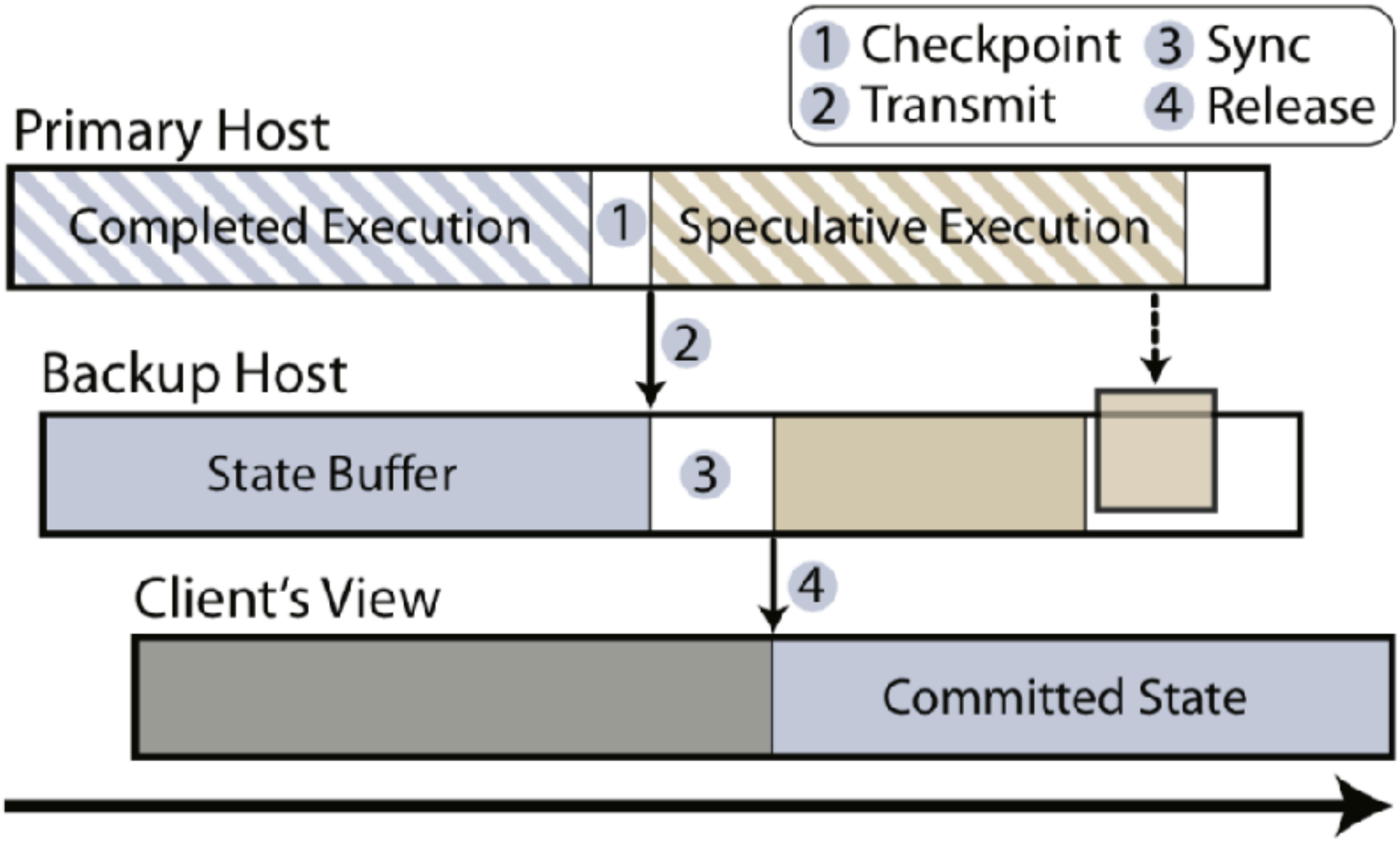


Figure 1: Speculative execution and asynchronous replication in Remus.

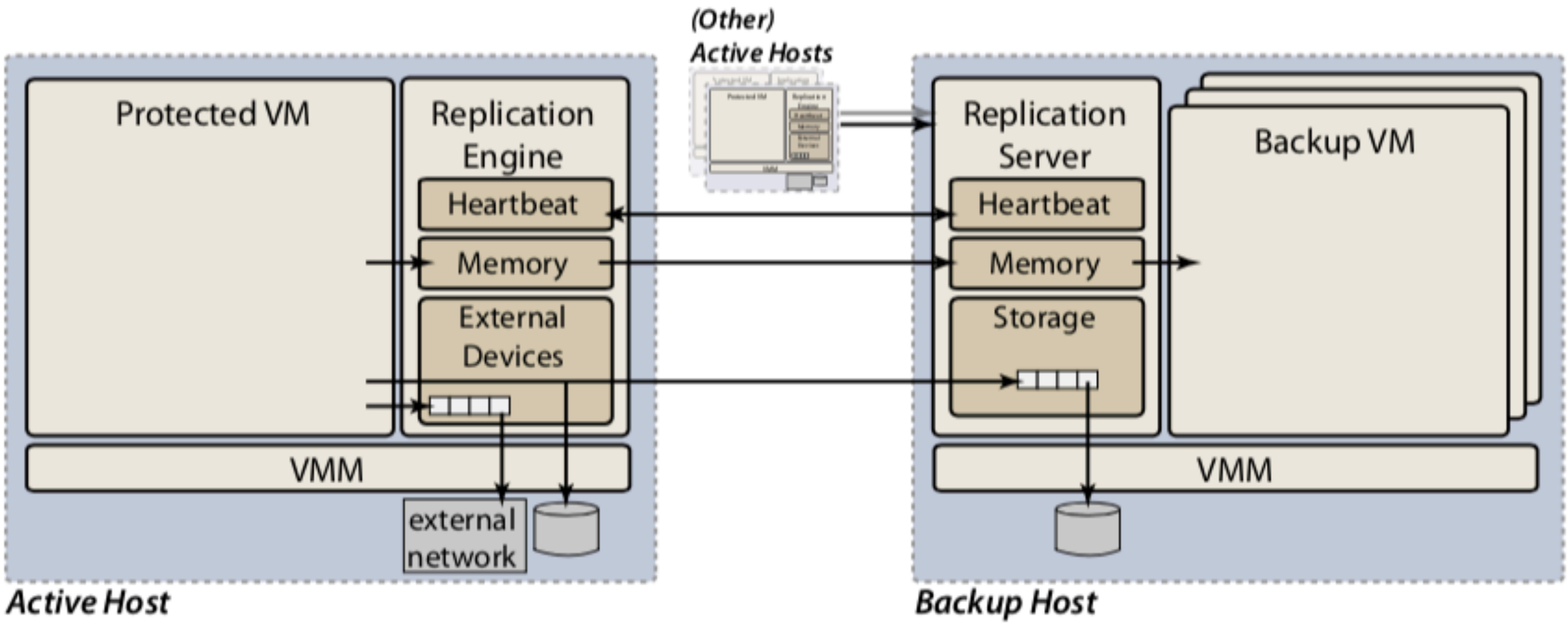


Figure 2: Remus: High-Level Architecture

Questions to think about

- How is snapshotting done efficiently?
  - Changes to xenstore
  - Making the copy fast
  - Reducing inter-process communication
- How is network traffic handled?
  - Inbound traffic delivered directly
  - Outbound traffic buffered until checkpoints
  - Use an intermediate queueing device

Handling disk writes

- Writes are buffered on backup node
- Writes are applied to backup storage at the end of the checkpoint
- This is done to ensure that the backup storage is consistent if there is a crash
- Only one of the two disk mirrors managed by Remus is valid at any given time; this is identified using an activation record

Feb 27
Impossibility Result and Consensus

Vijay Chidambaram

Consensus

- A set of asynchronous processes have to agree on a value
- In the more general case, each process proposes a value, and one of the values is agreed upon
- Simpler case is when the values are 0 or 1
- Some processes may fail
- Desired properties: termination, agreement, validity
- In FLP proof, we only need some process to decide (not all) - weak termination

Impossibility Result

- No completely asynchronous consensus protocol can tolerate even a single unannounced process death.
- Assumptions:
  - no byzantine failures (only fail-stop)
  - reliable communication channels (exactly once reliably delivery, not FIFO, may be delayed and re-ordered)
  - processes do not have access to a synchronized clock

Modeling the system

- Each process is a finite state automaton
- In one atomic step:
  - each process receives one message
  - does local computation in responses to message
  - sends out finite number of messages (atomic broadcast)

Window of vulnerability

- Period of time in commit or consensus protocol when the processes wait indefinitely for a failed process
- Impossibility result indicates every fully async commit or consensus protocol has this window of vulnerability

Modeling the system (more detail)

- A consensus protocol P is an asynchronous system of N processes (N >= 2)
- Each process p has a one-bit input register x_p, an output register y_p with values in (b, 0, 1), and an unbounded amount of internal storage
- Decision states: when output register has 0 or 1
- Output cannot change once decided

Modeling the system (more detail)

- Message format: (destination, message-value)
- send() and receive() similar to Project 1
- Configuration: internal state of each process + message buffers

- A consensus protocol is partially correct if it satisfies two conditions:
  - (1) No accessible configuration has more than one decision value.
  - (2) For each v E (0, I), some accessible configuration has decision value v.

Correctness Condition

- A consensus protocol P is totally correct in spite of one fault if it is partially correct, and every admissible run is a deciding run.
- Our main theorem shows that every partially correct protocol for the consensus problem has some admissible run that is not a deciding run.

Overview of proof

- The basic idea is to show circumstances under which the protocol remains forever indecisive.

- First, we argue that there is some initial configuration in which the decision is not already predetermined.

- Second, we construct an admissible run that avoids ever taking a step that would commit the system to a particular decision.

Step 1: There is a bivalent initial config

- Consider all initial configs: some lead to 0, some lead to 1
- Order all initial configs such that two configs which differ only in one value are next value
- Somewhere in this initial config, there must be a pair where one config is 0, another config is 1
  - These two configs differ only by the value of one process (lets call that p)
- We allow one process to fail, so 0-config should decide 0 even if p fails
  - But if it does not depend on p, 1-config (which only differs by p) should also decide 0.
  - Thus, 1-config is actually bivalent: it can decide 0 or 1

Step 2

- If you start from a bivalent config and apply message e, the resulting set of configs contains a bivalent config
- In other words, if you delay a message e long enough, you can move from a bivalent config to another bivalent config

Proving Step 2

- See https://www.the-paper-trail.org/post/2008-08-13-a-brief-tour-of-flp-impossibility/
- Good explanation of the proof
- Much more easier to understand than the paper

Terms

- C: starting bivalent config
- R: configs where e has not been received
- D: configs where e has been received

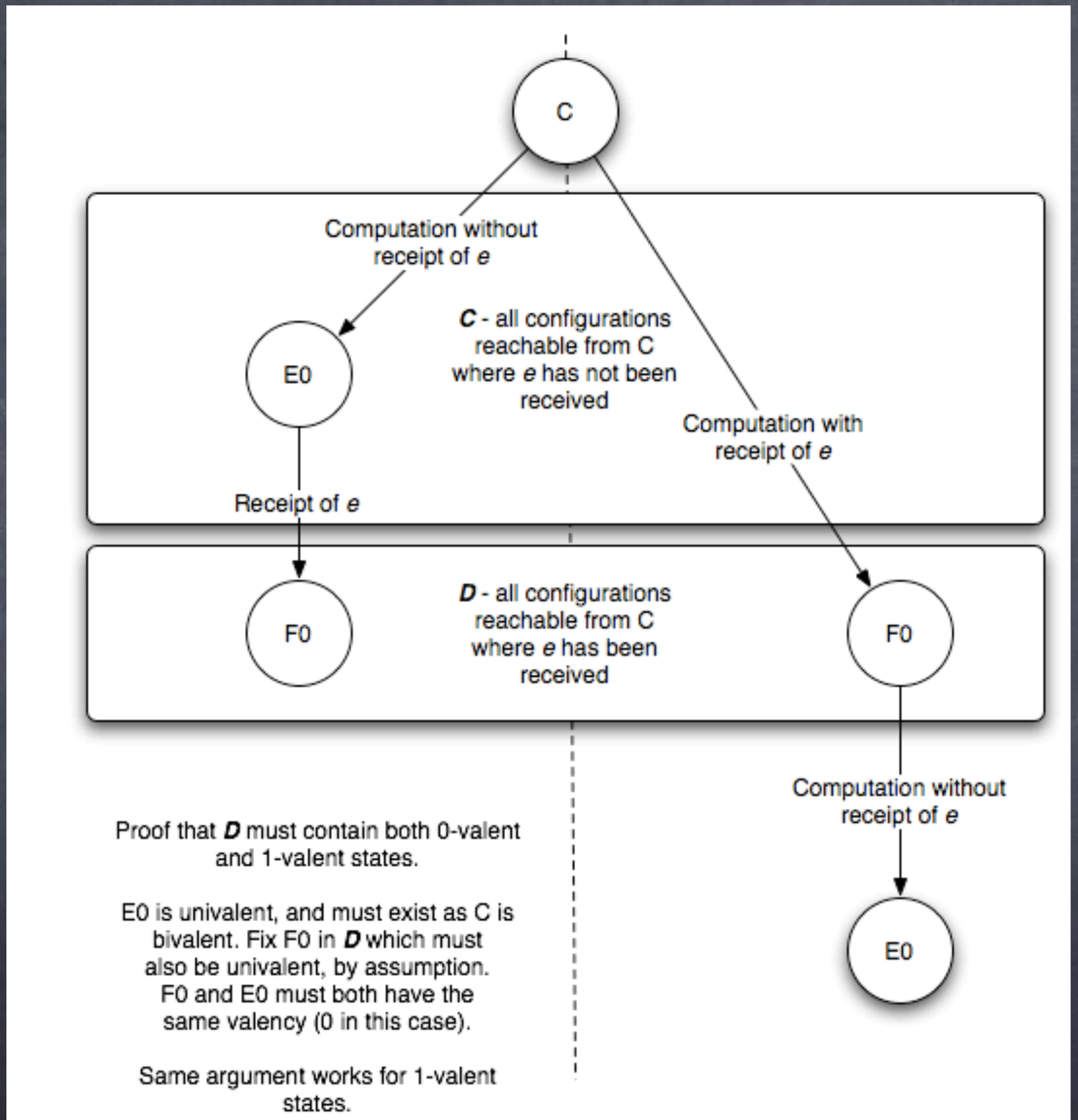
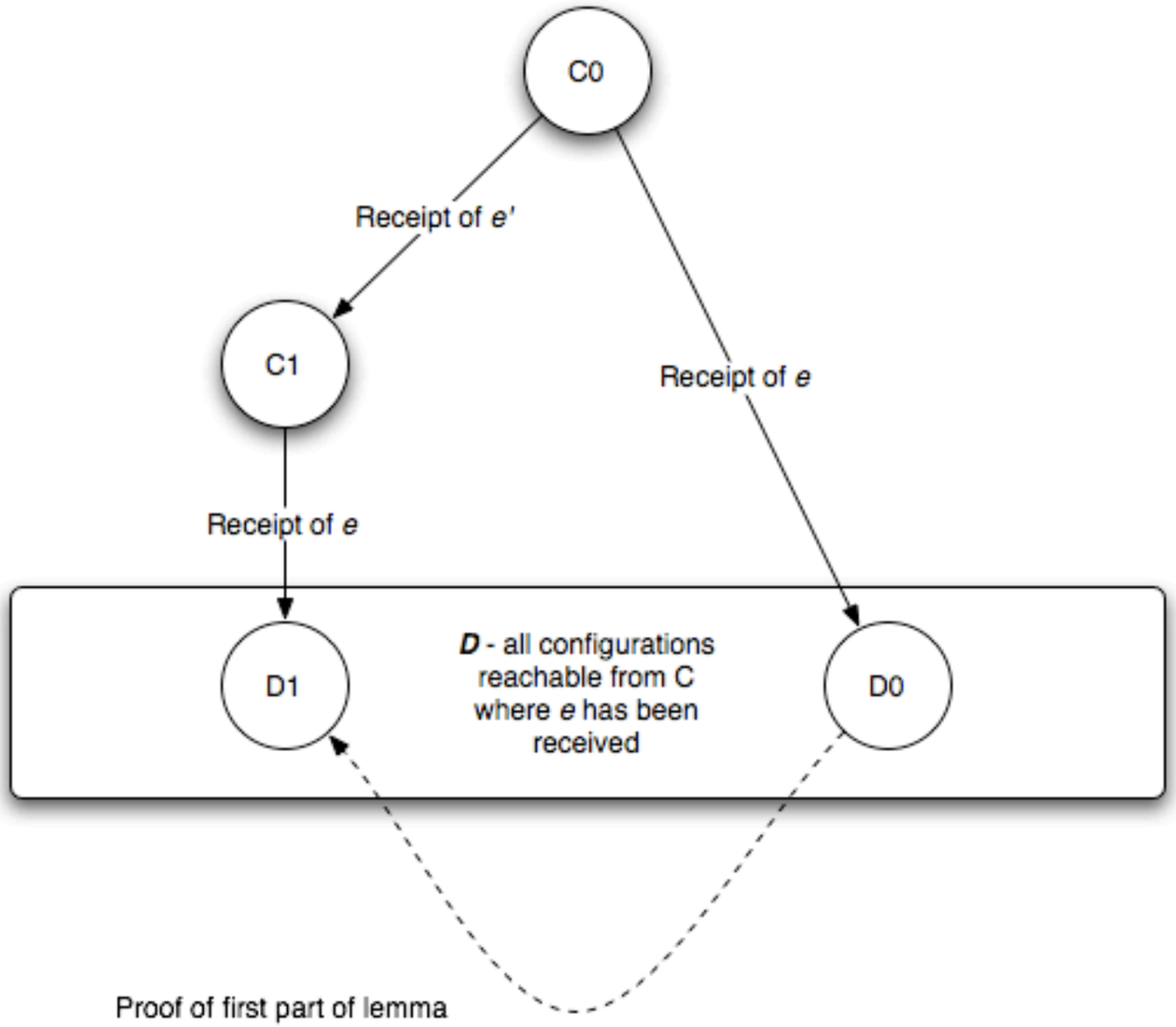

C
Computation without receipt of e
C - all configurations reachable from C where e has not been received
E0
Receipt of e
Computation with receipt of e
F0
D - all configurations reachable from C where e has been received
F0
Computation without receipt of e
E0
Proof that D must contain both 0-valent and 1-valent states.
E0 is univalent, and must exist as C is bivalent. Fix F0 in D which must also be univalent, by assumption. F0 and E0 must both have the same valency (0 in this case).
Same argument works for 1-valent states.

Figure 2.1

Established so far..

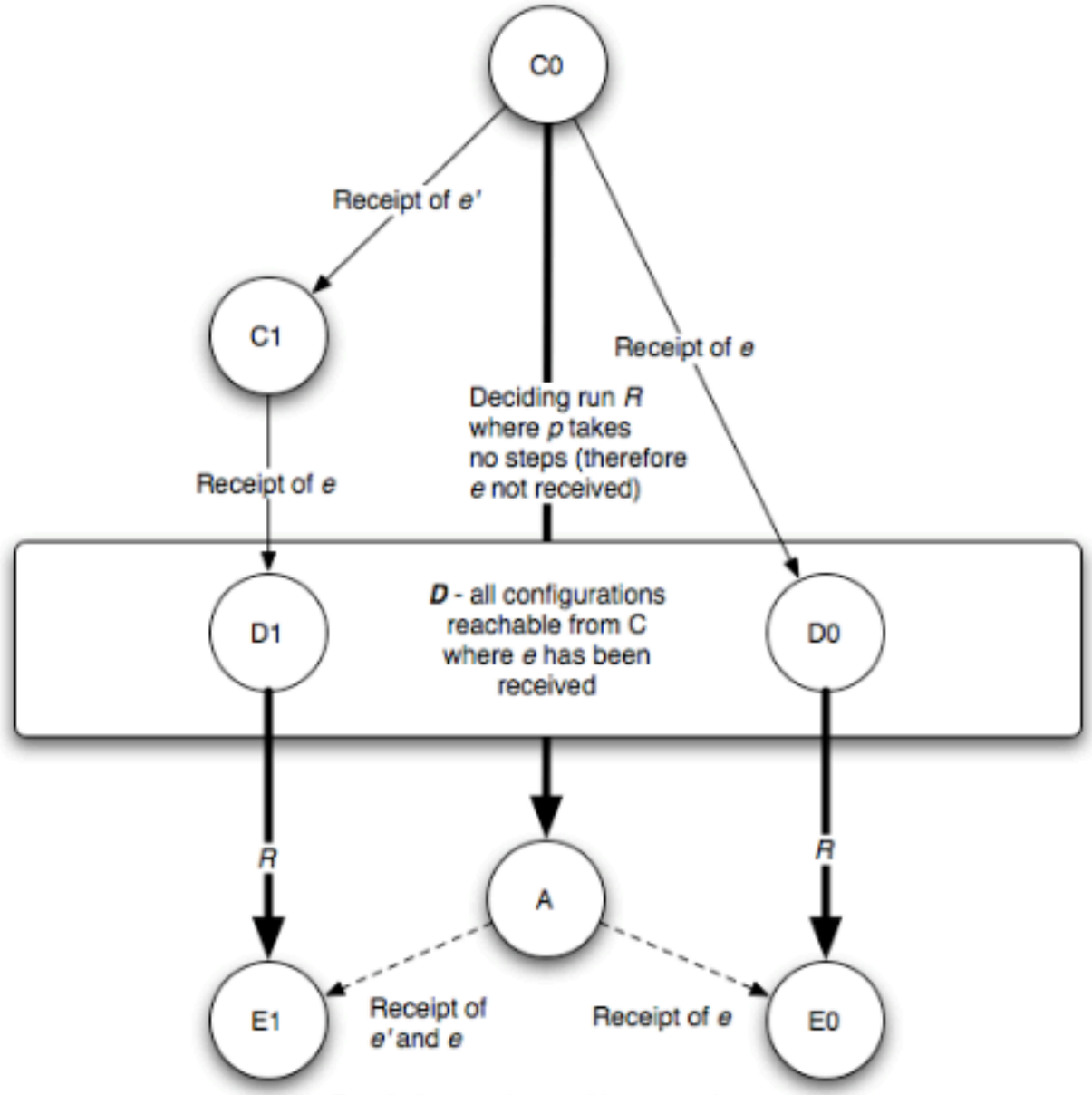
- That D must have both 0-configs and 1-configs
- Assumption: D does not have bivalent configs



Proof of first part of lemma that ***D*** must contain bivalent state.

By assumption, D0 and D1 are have different univalencies. However if messages *e* and *e'* don't go to the same process they may be applied in any ordering and all processes must end in same state (as they have seen the same order of messages locally).

This is a contradiction as now D0 is bivalent.



Proof of second part of lemma, where *e'* and *e* are addressed to the same process *p*: E0 and E1 are 0- and 1-valent respectively.

A is a univalent state reached by a deciding run from C0 where the process *p* for which *e'* and *e* are intended takes no steps (as if it had failed).

But at A either *e'* then *e* or just *e* can be received by *p* which places it in one of two univalent states. But A is itself univalent, and so this is a contradiction.

Constructing admissible runs which never decide

- We start from a bivalent configuration
- Let e be the message to be processed by process p next
- There is a bivalent configuration C' reachable from this config where e is the last message applied
- So we move from bivalent configs to other bivalent configs, and no decision is ever reached

Paxos

- P1. An acceptor must accept the first proposal that it receives.
  - An acceptor can accept multiple values
- P2. If a proposal with value v is chosen, then every higher-numbered proposal that is chosen has value v.
- P2a. If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor has value v.
- P2b. If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v.

Paxos

- P2c. For any v and n, if a proposal with value v and number n is issued, then there is a set S consisting of a majority of acceptors such that either (a) no acceptor in S has accepted any proposal numbered less than n, or (b) v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by the acceptors in S.

- P1a . An acceptor can accept a proposal numbered n iff it has not responded to a prepare request having a number greater than n.

The Paxos Algorithm

- Phase 1. (a) A proposer selects a proposal number n and sends a prepare request with number n to a majority of acceptors.
- (b) If an acceptor receives a prepare request with number n greater than that of any prepare request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal (if any) that it has accepted.

The Paxos Algorithm

- Phase 2. (a) If the proposer receives a response to its prepare requests (numbered n) from a majority of acceptors, then it sends an accept request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.

- (b) If an acceptor receives an accept request for a proposal numbered n, it accepts the proposal unless it has already responded to a prepare request having a number greater than n.

Mar 3
Paxos Algo, Paxos Made Live

Vijay Chidambaram

Paxos Made Live

- Actual engineering challenges in deploying Paxos
- From Chubby developers at Google
- Chubby provides locking and small-file storage for Google File System, BigTable, etc
- Chubby used as a metadata server in most cases

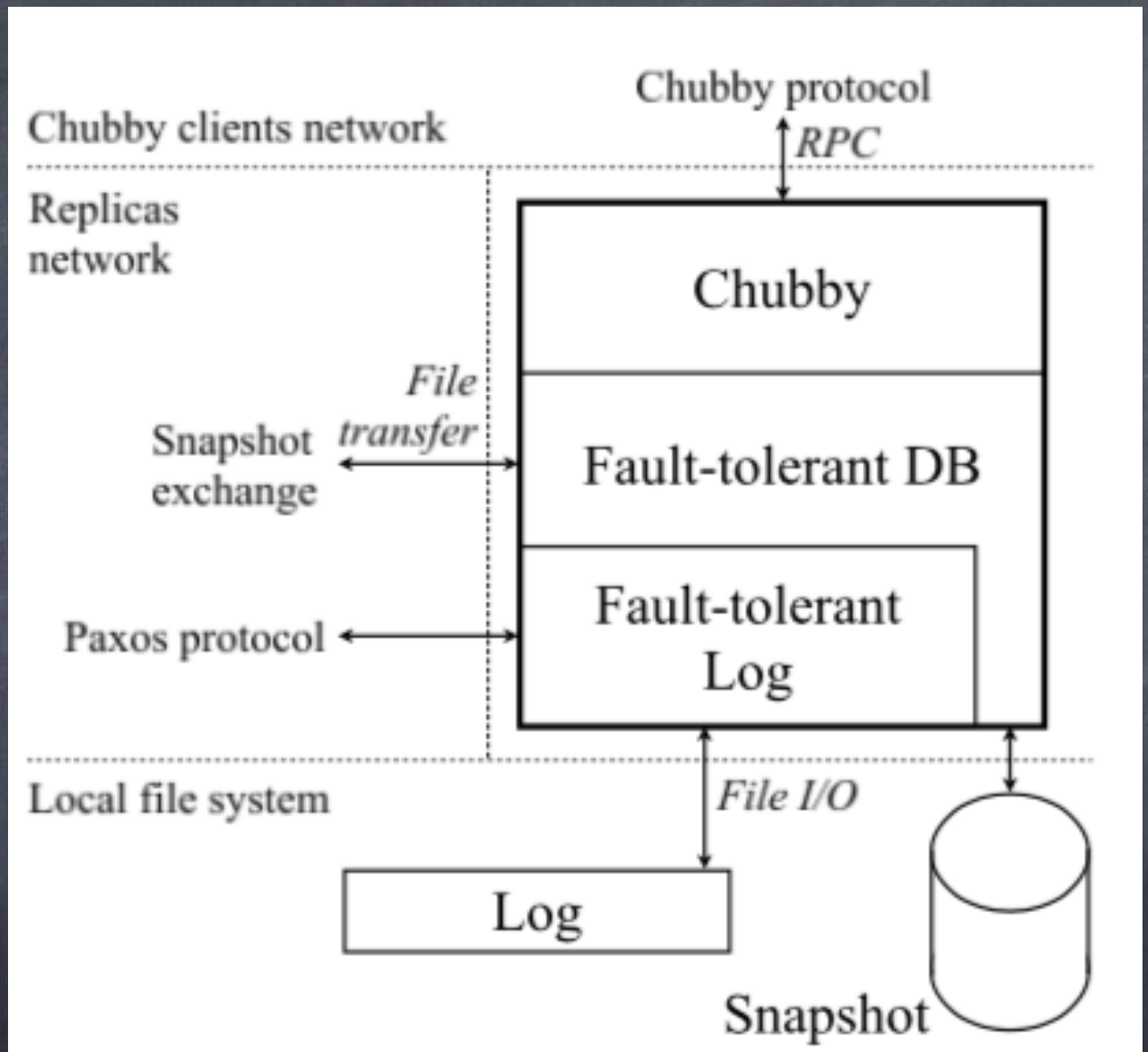


Figure 1: A single Chubby replica.

Deployment Issues

- Handling disk corruption
- Master leases
- Epoch numbers
- Snapshots

Database Transactions

- Interesting way to do transactions
- MultiOp:
  - List of conditions
  - An action to be performed if the condition is true
  - An action to be performed if the condition if false
  - Each db action applies to a single row or data item

Expressing the algorithm

- Designed a simple specification language
- This language is compiled down into C++
- In the spec language, Paxos can be expressed much more briefly

Testing

- Testing safety and liveness
  - Testing safety first
  - Allow system to recover
  - Then test liveness

Mar 5
Viewstamped Replication

Vijay Chidambaram

Viewstamped Replication

- Replication technique that handles node failures
- Provides consensus among replicas
- Alternative to Paxos, the protocol behind RAFT
- Slightly different goal: replication versus consensus
- Viewstamped Replication uses consensus internally

Assumptions

- Failures are fail-stop
- Does not handled Byzantine failures
- Works in an async environment, similar to Paxos
- Assumes only a minority of replicas fail. To tolerate f failures, total 2f + 1 nodes required

Quorum Intersection Property

- Quorum: f + 1 replicas
- All steps in the protocol executed by a quorum
- Quorum intersection property: any two quorums must intersect in at least one node
- We saw this in Paxos as well

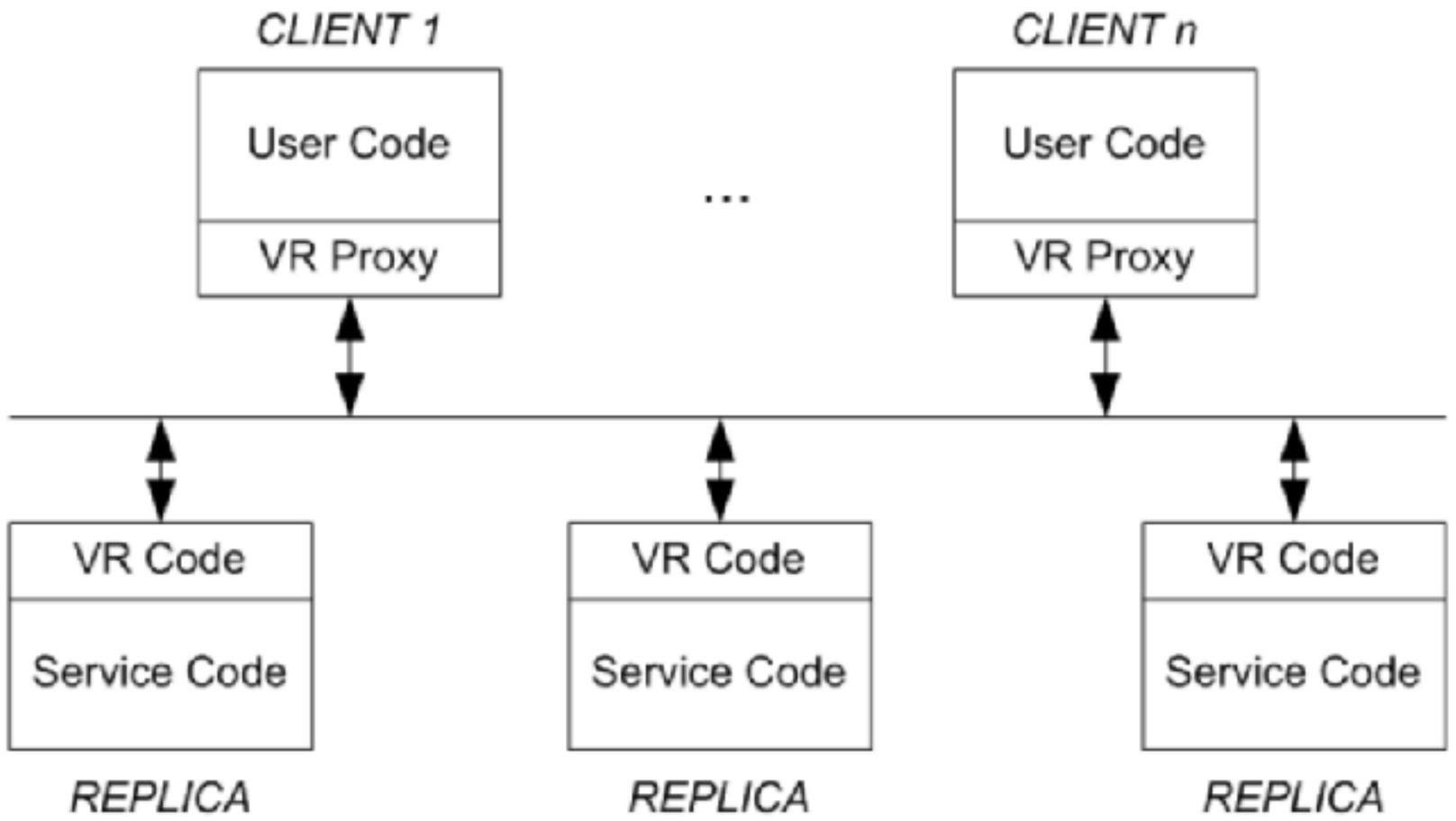


Figure 1: VR Architecture; the figure shows the configuration when $f = 1$.

Performing Updates

- Updates are serialized and performed by a primary replica
- The order determined by the primary is used by the backup replicas
- The system moves through "views": each view has a primary replica
- If the primary fails, a view change protocol selects a new primary

View Changes

- A view change should not result in old updates being lost
- A view change is processed by a quorum
- Each request is also handled by a quorum
- Hence there will be at least one node that remembers the update before the view change

Recovering from failure

- A replica that recovers from a transient failure can rejoin the cluster
- Can only rejoin when its state is equivalent to its state at the failure point
  - VR does not keep disks, which would trivially satisfy this requirement

Selecting the primary

- Primary selected in round robin fashion
- Replica i is the primary in view (i mod n)
- Given view number and number of replicas, primary can be computed

- The *configuration*. This is a sorted array containing the IP addresses of each of the $2f + 1$ replicas.
- The *replica number*. This is the index into the configuration where this replica's IP address is stored.
- The current *view-number*, initially 0.
- The current *status*, either *normal*, *view-change*, or *recovering*.
- The *op-number* assigned to the most recently received request, initially 0.
- The *log*. This is an array containing *op-number* entries. The entries contain the requests that have been received so far in their assigned order.
- The *commit-number* is the *op-number* of the most recently committed operation.
- The *client-table*. This records for each client the number of its most recent request, plus, if the request has been executed, the result sent for that request.

Figure 2: VR state at a replica.

Normal Operation

- Condition: view number of node has to match view number of request
- Client -> Primary REQUEST
- Primary -> Replicas PREPARE
- Replicas -> Primary PREPARE OK
- Primary -> Replicas COMMIT

View Changes

- Replica -> Others STARTVIEWCHANGE (view++)
- Replicas -> New Primary DOVIEWCHANGE with log
- New Primary uses longest log in new view
- New Primary -> Replicas STARTVIEW

Recovery

- Recovering node (RN) -> Replicas RECOVERY
- Replicas -> RN RECOVERYRESPONSE, with logs
- RN updates its log based on response from Primary

Optimizations

- Fast reads at the primary
- Witnesses
- Batching

Reconfiguration

- PREPARE with new config
- Increment epoch number
- COMMIT to old replicas that are still in config
- STARTEPOCH to new replicas that are being added
- Nothing to old replicas not part of new config
- Important: while transitioning, no new requests are accepted

Reconfig: members of new group

- STARTEPOCH or COMMIT message
- Records old and new configs
- New epoch-number, view 0
- status: transitioning
- catches up state is required
- status: normal
- EPOCHSTARTED to replicas that are being replaced

Reconfig: members being replaced

- COMMIT message
- Records old and new configs
- New epoch-number, view 0
- status: transitioning
- waits for f+1 EPOCHSTARTED
- shuts down