CMPE 282 Cloud Services
Concepts: CAP, Paxos

Instructor: Kong Li

Content

- Concurrency
- Distributed Systems
- Process, communication channel, event
- Time, Logical Clocks
- Message delivery
- CAP
- CAP 12 Years Later
- Paxos
- Paxos Advanced

Concurrency

- Issues
  - race conditions: result depends on sequences of events
  - Shared resources: locks, semaphores, monitors
  - Deadlock: blocking and no progress

Livelock: non-blocking and no progress

- The four *Coffman conditions* must hold simultaneously for a deadlock to occur:

Single instance resource type: necessary and sufficient for deadlock
Multiple instance resource type: necessary, not sufficient, for deadlock

  - Mutual exclusion: at least one non-sharable resource held exclusively
  - Hold and wait: at least one holds resources and waits for others
  - No-preemption: cannot force one to release lock
  - Circular wait in wait-for graph
- Priority inversion: a higher priority proc/task is indirectly preempted by a lower priority one
  - Mars Pathfinder http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html

Distributed Systems

- Def: autonomous computers + network + distribution software (middleware)
  - enables computers to coordinate activities and share resources
- Characteristics:
  - A single/integrated computing facility
  - autonomous components
  - Separate scheduling, resource management, security policies on each system
  - Multiple points of control and failure
  - resources may not be accessible at all times
  - Scalability
  - Availability

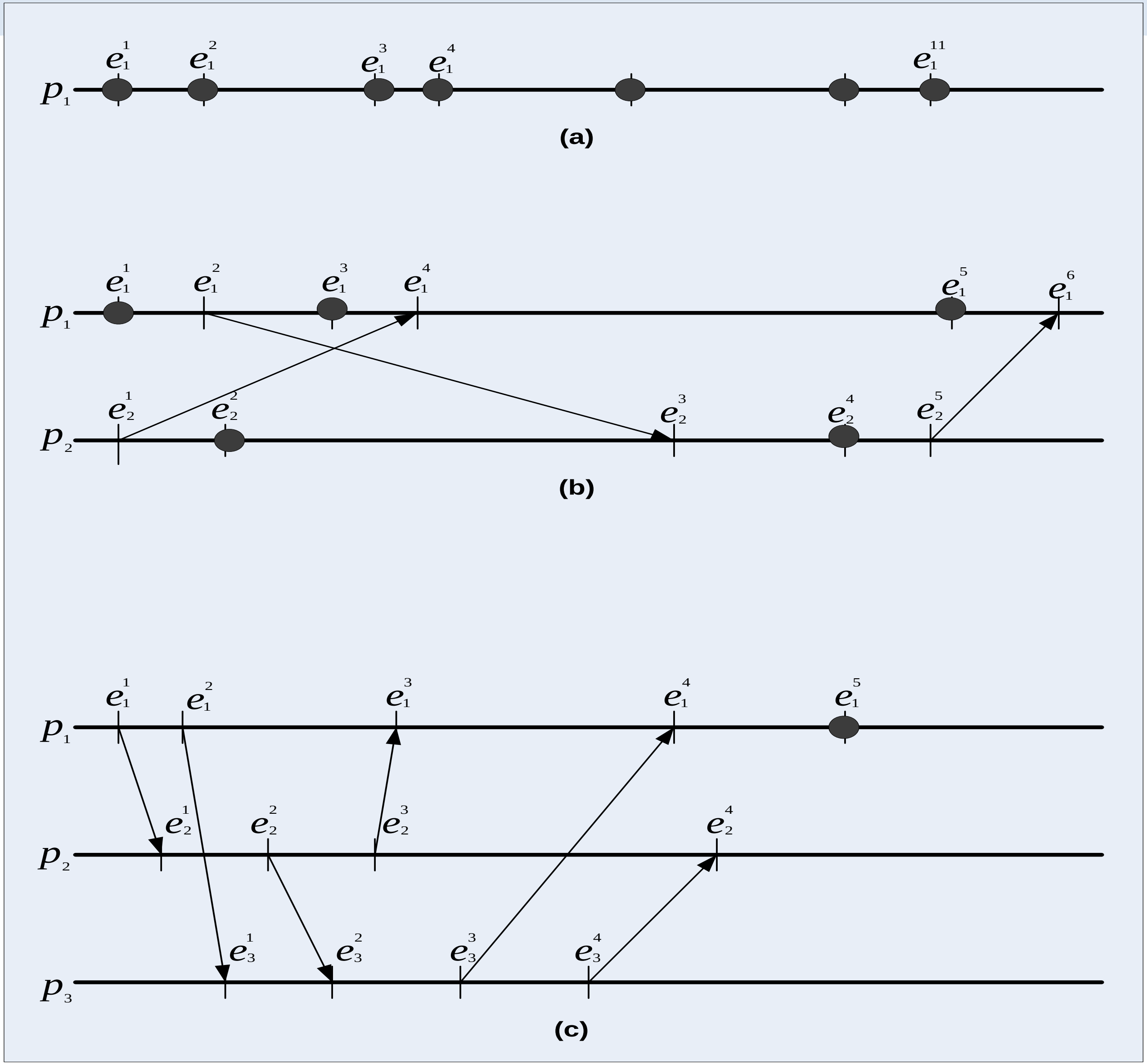
Distributed System: Desirable Properties

- Access transparency: identical operation for local and remote objects
- Location transparency: objects accessed w/o knowing their locations
- Concurrency transparency: concurrent processes using shared information objects without interference among them
- Replication transparency: objects replication w/o user's knowledge
- Failure transparency
- Migration transparency: objects migration w/o affecting operation performed on them
- Performance transparency: reconfigured based on the load and QoS
- Scaling transparency: scale without changing system structure and applications

Process and Communication Channel

- Process
  - State of process/thread
  - Event: change of state of a process
    - Local events
    - Communication events
  - Process group: a set of cooperating / communicating processes
- Communication channels
  - Message
  - *send(m)* and *receive(m)* communication events, where *m* is a message
  - Assumptions: Uni-directional, unreliable
  - State of the channel $\xi_{i,j}$ from p_i to p_j: consists of messages sent by p_i but not yet received by p_j
  - Protocol
- Global state of a distributed system
  - union of the states of the individual processes and channels

Local and Communication Events



Time and Time Intervals

- History: a sequence of events, each corresponds to a change of the state
  - Local history
  - How to collect global history? Need certain coordination
- Process coordination requires:
  - A global concept of time shared by cooperating entities
  - The measurement of time intervals
- Local timers: relative time measurements
- Global agreement on time
  - Precedence relation, temporal ordering, event causality
- Timestamps: used for event ordering using a global time base constructed on local virtual clocks

Time and Time Intervals (cont'd)

- w/o global time, events can be ordered only based on cause-and-effect
  - Cause-effect relationship: cause must precede effects
  - $e \rightarrow e'$ means e' may have been influenced by event e
- e^k_i: k-th event in the local history h_i of process p_i
  - Causality of local events

    If $e^k_i, e^l_i \in h_i$ and $k < l$ then $e^k_i \rightarrow e^l_i$
  - Causality of communication events between p_i and p_j

    If $e^k_i = send(m)$ and $e^l_j = receive(m)$ then $e^k_i \rightarrow e^l_j$
  - Transitivity of the causal relationship

    If $e^k_i \rightarrow e^l_j$ and $e^l_j \rightarrow e^n_m$ then $e^k_i \rightarrow e^n_m$
- Concurrent events: neither one caused the other

Logical Clocks

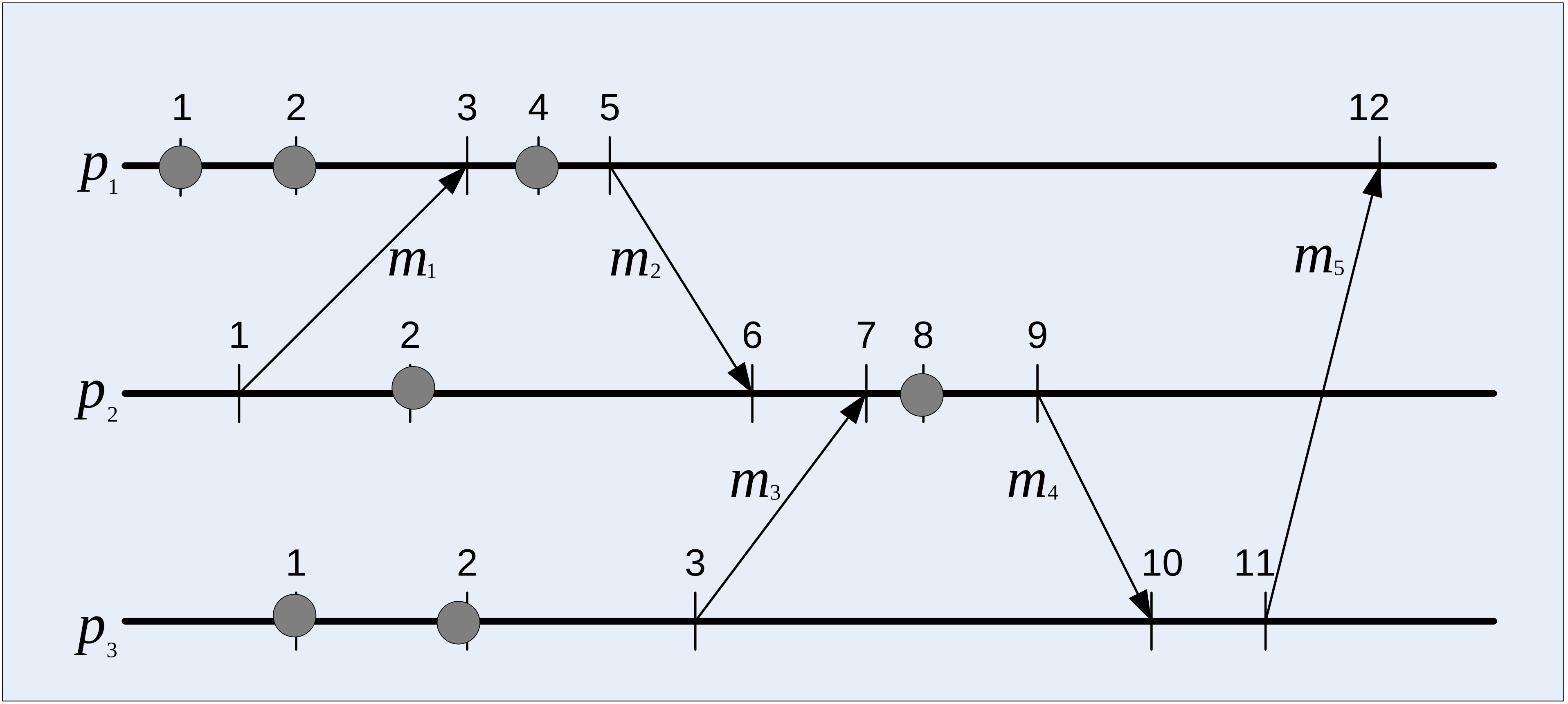
- Logical clock (LC): an abstraction to ensure the clock condition in the absence of a global clock
  - A process maps events to positive integers
  - LC: current logical clock value
  - LC(e): the local variable associated with event e
- TS(m): time stamp of each msg m
  - Logical clock at the time of sending
  - TS(m) = LC(*send*(m))
- Rules to update the logical clock:

  LC(e) = LC + 1 if e is a local or *send(m)* event

  LC(e) = max(LC, TS(m)) + 1 if e is a *receive(m)* event

- If $e \rightarrow e'$ then LC(e) < LC(e')

Logical Clocks (cont'd)



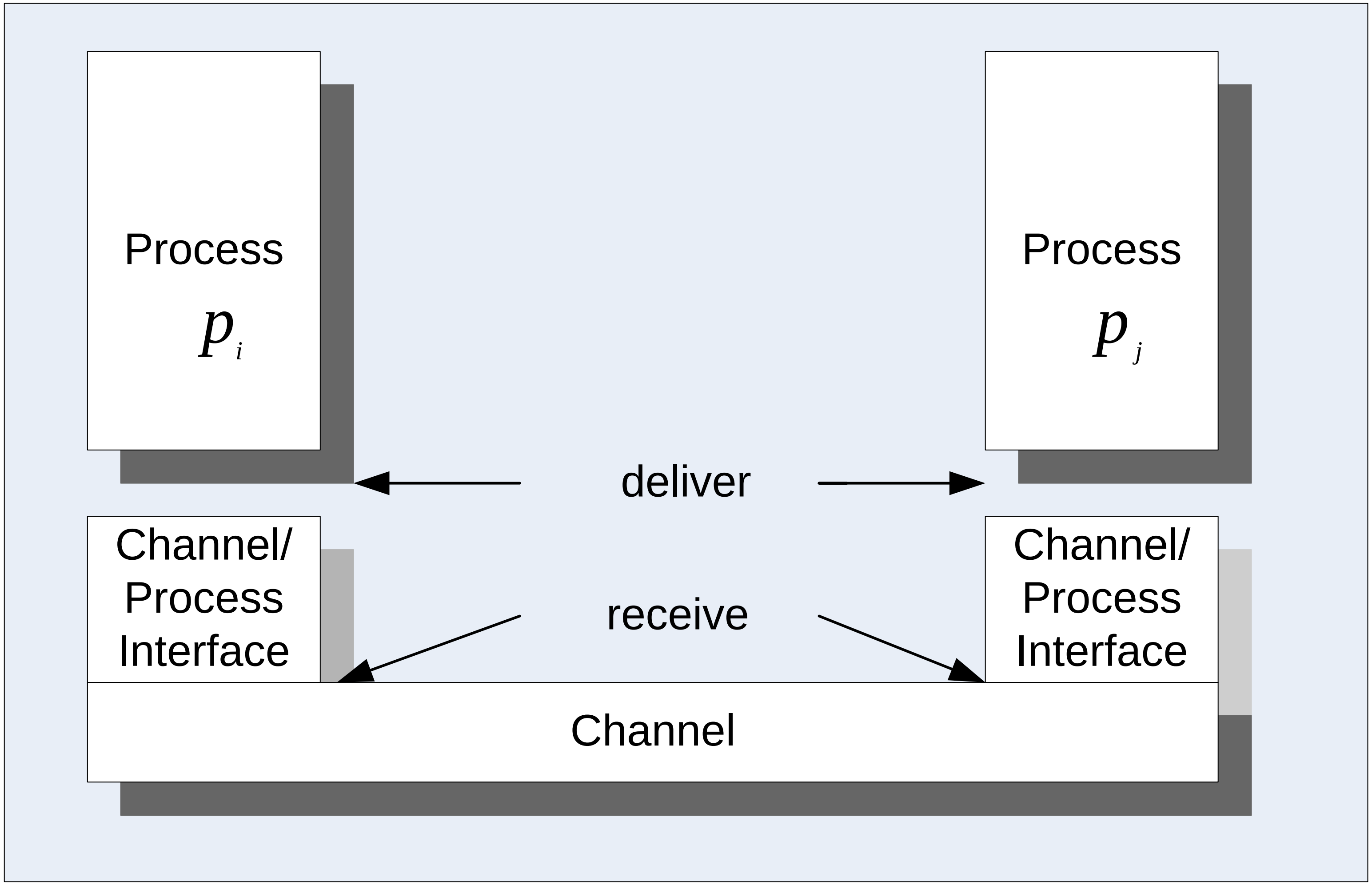
Three processes and their logical clocks (only LC values are shown)

- Only partial order is possible; global order is not possible
- No gap detection, i.e. determine if *e'* exists such that LC(e^3_3) < LC(e') < LC(e^4_3)

Message Delivery

- Communication channel: no assumptions about the msg order
  - real-life network might reorder msgs
- Msg receiving and msg delivery: two distinct operations
  - receive(m) → deliver(m)
- First-In-First-Out (FIFO) delivery
  - Msgs delivered in the same order they are sent
  - $send_i(m) \rightarrow send_i(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$
- Causal delivery: extension to FIFO + recv msg from multi sources
  - When p_i sends m to p_k, and p_j sends m' to p_k

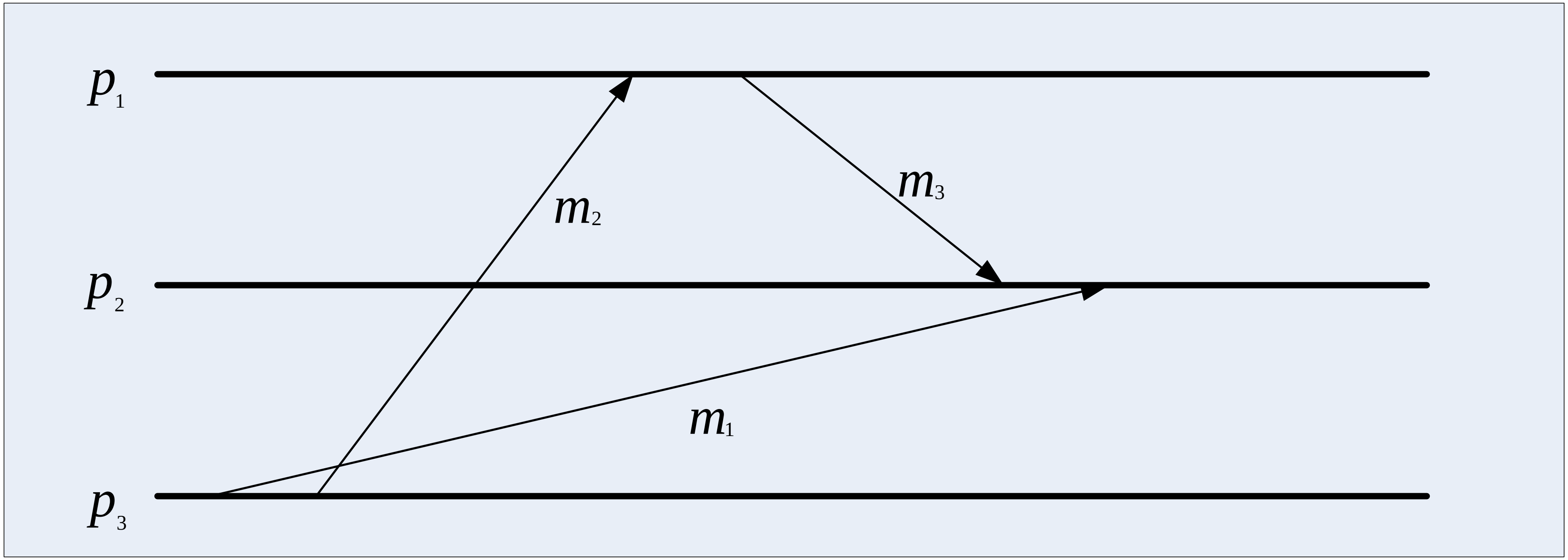
    $send_i(m) \rightarrow send_j(m') \Rightarrow deliver_k(m) \rightarrow deliver_k(m')$
  - Know the entire system using only local info
- If channel is not FIFO, FIFO delivery can be enforced by attaching sequence # to each msg sent
  - sequence # also used to reassemble msgs out of individual packets



Msg receiving and Msg delivery are two distinct operations. The channel-process interface implements the delivery rules, e.g., FIFO delivery.

Message Delivery (cont'd)

- FIFO does not guarantee causal delivery!
  - FIFO between each pair but may violating causal delivery; msg m_1 is delivered to process p_2 after msg m_3, though msg m_1 was sent before m_3



Replication with Weak Consistency

- Why data replication?
- Data replication w/ weak degrees of consistency (w/o serializabiliy)
  - Usually only when not enough sites are available to ensure quorum
  - Tradeoffs: consistency vs availability or latency/performance
- Key issues:
  - Reads may get old versions
  - Writes may occur in parallel, leading to inconsistent versions
  - Q: how to detect, and how to resolve? A: Version vector scheme
- Primary-slave replication
  - Read from any site
  - Updates done at a single "master" site, and async propagated to "slave" sites
- Multi-master replication
  - updates are permitted at any replica; system propagates to replicas *transparently*
- Lazy replication
  - Updates to replicas are transmitted *after* transaction commits
  - Allows updates to occur even if some sites are disconnected

Tradeoffs?

Brewer's CAP Theorem

- Three desirable properties when designing distributed sys
  - **C**onsistency: All copies have the same value
  - **A**vailability: System can run even if parts have failed
  - **P**artition-tolerant: Survive network partitioning
- It is *impossible* to achieve all three in
  - Async networks: no clock; node makes decision based on msg received and local computation
  - Partially sync networks: each node has local clock; all clocks increase at the same rate; clocks are not synchronized
- Any two of these three can be achieved
- Examples
- Large systems will partition at some point → availability or consistency?
  - Traditional DB chooses consistency
  - Most web apps choose availability (exception: order processing, etc.)
- CAP theorem only matters when there is a partition

Eventual Consistency

- Eventual consistency: when no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent
  - For a given accepted update and a given node, eventually either the update reaches the node or the node is removed from service
  - You may not know how long it may take
- Known as **BASE** (**B**asically **A**vailable, **S**oft state, **E**ventual consistency), as opposed to ACID
  - **Soft state**: copies of a data item may be inconsistent
  - **Eventually Consistent** – copies becomes consistent at some later time if there are no more updates to that data item
- Used by most NoSQL
- Tradeoffs: availability, consistency, latency/performance

CAP 12 Years Later

- What if the system is not partitioned?
  - No tradeoff needed while connected
- Partition detection: disagreement
- Partition exists and in the same system, the choice b/w C & A
  - How many times can it occur?
  - can make different choices at different times
    - the choice can change according to the operation or data or user
- All three properties are more continuous than binary
  - Different levels of availability, consistency, partition, etc.
  - Two options in partition mode:
    - Record extra information (→ more availability)
    - Limit operations (→ more consistency)
  - CAP Decisions at fine granularity!
- Recovery: Re-establishing consistency, compensating errors

Consensus Protocols

- Consensus: process of agreeing to one of several alternates proposed by a number of agents
- Distributed consensus problem
  - Group of processes must agree on a single value
  - Value must be proposed
  - After value is agreed upon, it can be learned
  - Non-blocking
  - Deadlock free
- Assumptions:
  - Processes run on processors and communicate through a network
  - processors and network may experience failures
  - Processors operate at arbitrary speed; have stable storage; may rejoin the protocol after a failure; and can send msg to any other processor
  - The network may lose, reorder, or duplicate msg; may take arbitrary long time to deliver the msg (msg cannot be corrupted)

Paxos

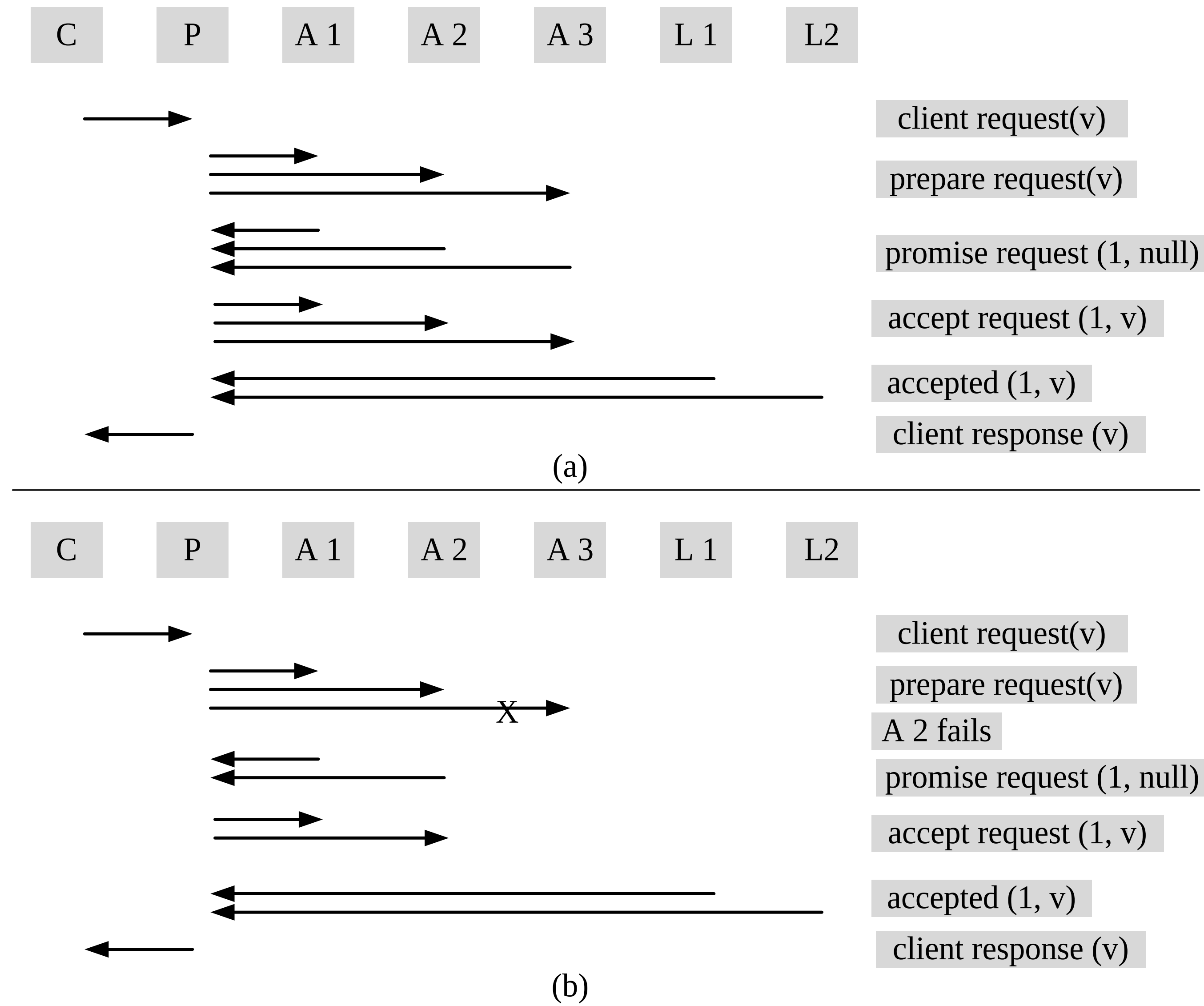
- Paxos: a family of non-blocking protocols to reach consensus in distributed system based on a deterministic state machine
- Types of entities in basic Paxos:
  - Client: agent that issues a request and waits for a response
  - Proposer: agent to advocate a request from a client, and to act as a coordinator to move the protocol forward in case of conflicts
  - Acceptor: agent acting as the *fault-tolerant memory* of the protocol
  - Learner: agent acting as the replication factor of the protocol and taking action once a request has been agreed upon
  - A single process may act as more than one agent
- A proposal has a proposal number *pn* and (in 2nd phase) contains a value *v*
- Several types of requests flow through the system, *prepare, accept*

The Paxos Algorithm

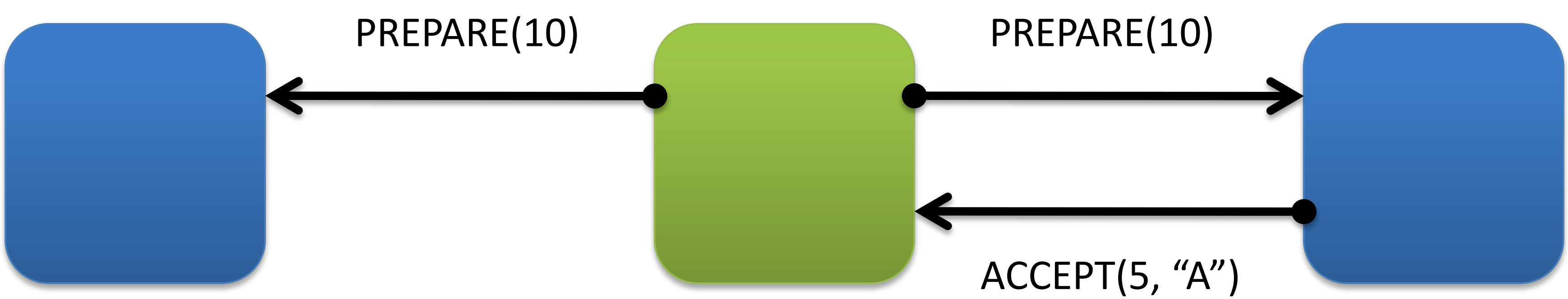
- **Phase I**
  - Proposal preparation: A proposer selects a proposal number *n* and sends a *prepare request* with number *n* to a majority of acceptors
  - Proposal promise: If an acceptor receives a *prepare request* with number *n* greater than that of any *prepare request* to which it has already 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 less than *n* (if any) that it has accepted, else it does not reply
- **Phase II**
  - Accept request: If the proposer receives a response to its *prepare request*s (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
  - Accept: 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*

The basic Paxos with three actors: proposer (P), three acceptors (A1,A2,A3) and two learners (L1, L2). The client (C) sends a request to one of the actors playing the role of a proposer. The entities involved are (a) Successful first round when there are no failures.

(b) Successful first round of Paxos when an acceptor fails.



Example: Prepare

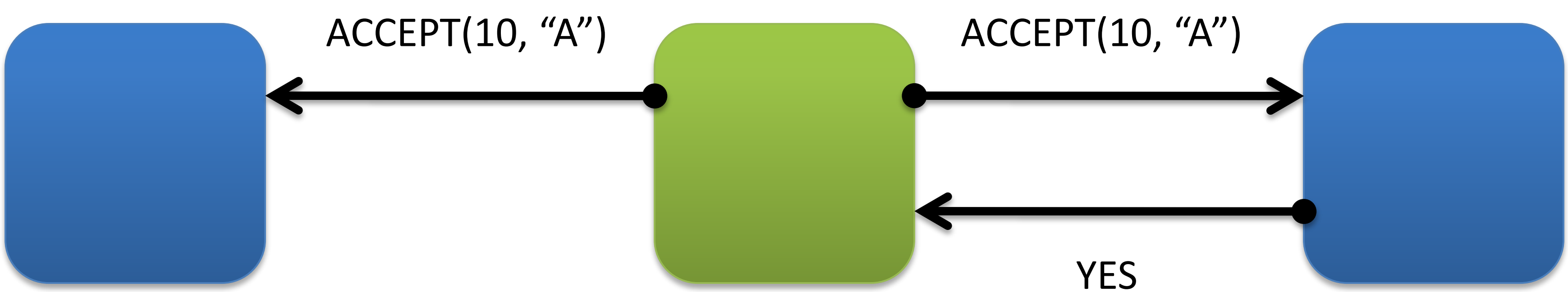


Highest Accept:
(5, “A”)
Highest Prepare:
(15)

Highest Accept:
(5, “A”)
Highest Prepare:
(~~8~~10)

http://www.cs.berkeley.edu/~istoica/classes/cs294/11/notes/07-gene-paxos.pptx

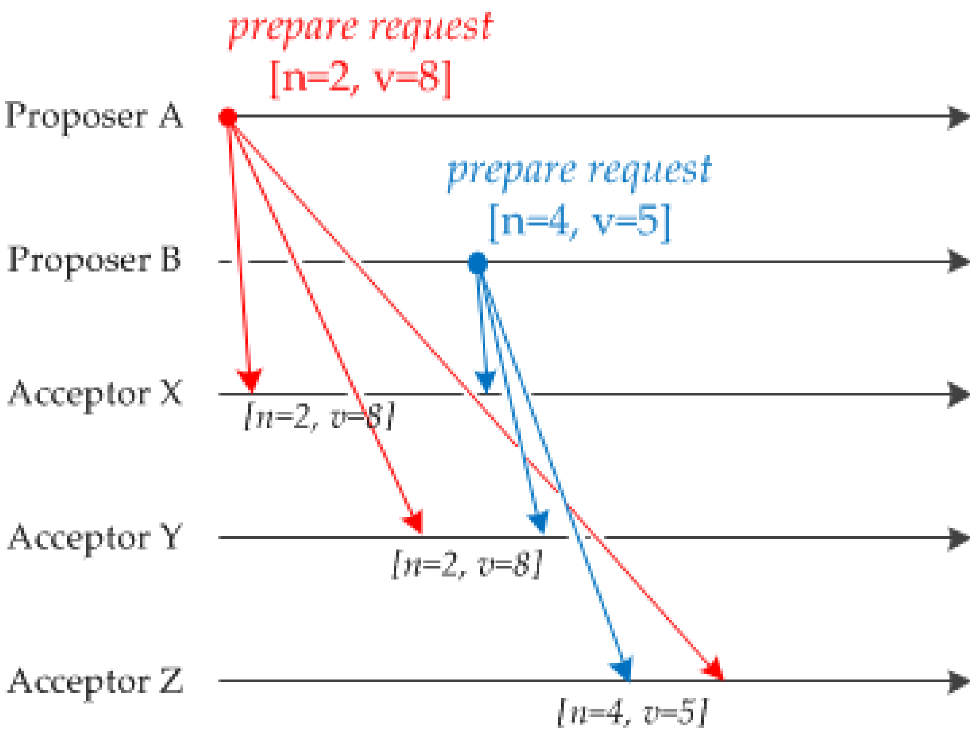
Example: Accept



Highest Accept:
(5, “A”)
Highest Prepare:
(15)

Highest Accept:
(10, “A”)
Highest Prepare:
(10)

Example 2: phase 1 proposal



Ignore the value v in *prepare request*

- https://angus.nyc/2012/paxos-by-example/
- Proposers A and B each send prepare req to every acceptor
- Proposer A's req reaches acceptors X and Y first, and proposer B's req reaches acceptor Z first

Paxos: Implications

- Proposer
  - can make multiple proposals, so long as it follows the algorithm for each one
  - can have > 1 proposers; proposal number is globally unique and monotonically increasing
  - can abandon a proposal in the middle of the protocol at any time
    - abandon a proposal if some proposer has begun to issuing a higher-numbered one
- Acceptor
  - can ignore any proposal or accept requests without compromising safety
  - can also ignore prepare requests for proposals it has already accepted
- Learner
  - The acceptors could respond with their acceptances to some set of *distinguished learners*, each of which can then inform all the learners when a value has been chose
- Paxos implementation: deterministic state machine
  - The state machine has a current state
  - The state machine performs a step by taking as input a command and producing an output and a new state

Paxos: Properties

- Non-blocking as long as a majority of participants are alive
  - vs 2PC (blocking when coordinator failed after prepare msg is sent, until it recovers)
  - Value is chosen only in Phase 2 when majority accepts
    - the value from the highest-numbered response (acceptor has accepted)
- SAFETY

  Not involved w/ definition nor protocol

  - Only a value that has been proposed can be chosen
  - Only a single value is chosen
  - A process never learns that a value has been chosen unless it has been
- LIVENESS

  Not involved w/ definition nor protocol

  - Some proposed value is eventually chosen
  - If a value is chosen, a process eventually learns it

  no deterministic fault-tolerant consensus protocol can guarantee progress in an asynchronous network

- Ex
  - ZooKeepr coordination service: zookeeper.apache.org
  - Google: Chubby locking service (GFS, BigTable, Megastore), Spanner
  - Windows Azure storage, Cosmos DB

Paxos Applications

- leader election
- fault-tolerant replicated state machine
  - Storage or db replication: all replicas in same state given the same input sequence
    - Google Megastore/Spanner, Microsoft Azure storage
  - distributed transactions (all replicas execute/log the same op)
  - distributed file server (agree on the same session id)

Paxos: advanced

- Simple / common case: one proposer
  - Must accept proposal, otherwise no progress
- *P1: Acceptor must accept the first proposal it receives*
- Multiple proposers?
  - Proposer 1 sends value v to N/2 acceptors
  - Proposer 2 sends value v' to the other N/2 acceptors
  - P1 → each acceptor accepts the corresponding proposal – no majority
- Solution: acceptors must accept multiple proposals
  - Each proposal has a unique proposal number
  - *Multiple* proposals may be chosen (accepted by a majority)

Paxos: advanced (cont'd)

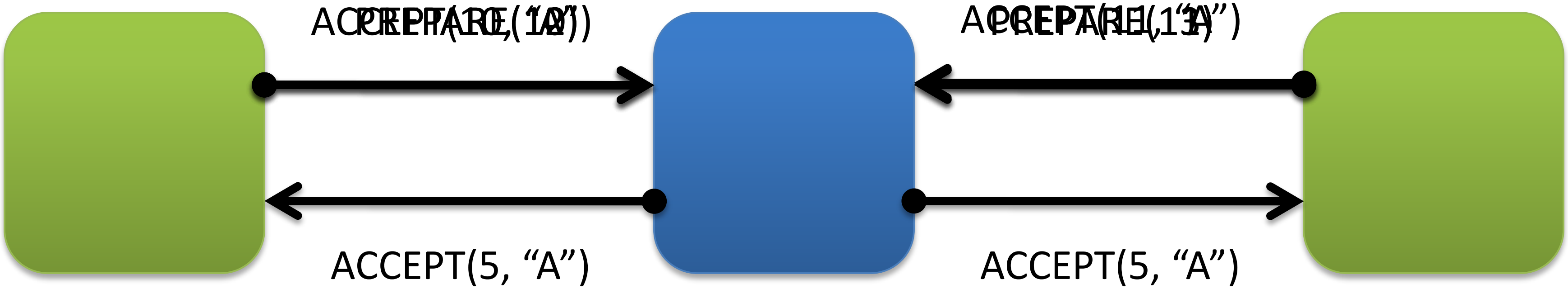
- A chosen proposal is accepted by majority of acceptors
- Proposer must know what proposals have been accepted by a majority of acceptors
- *P2. If a proposal with value v is chosen, then every higher-numbered proposal that is **chosen** has value v*
  - *$P2^a$. If a proposal with value v is chosen, then every higher-numbered proposal **accepted** by any acceptor has value v*
  - *$P2^b$. If a proposal with value v is chosen, then every higher-numbered proposal **issued** by any proposer has value v*
  - *$P2^c$. 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:*
    - *no acceptor in S has accepted any proposal numbered less than n, OR*
    - *v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by acceptors in S*

Paxos: advanced (cont'd)

- What if
  - p with proposal number m asks c about accepted proposals
  - c replies with empty set {}
  - p' proposes (n, v) to c, where n < m
  - c accepts p'
  - p proposes (m, v') to c, with m > n - violating $P2^c$
- Solution: do not accept p'
  - p asks c about accepted proposals with numbers less than m *(prepare request)*
  - c replies with empty set {}, and promises not to accept proposals with numbers less than m
  - p' proposes (n, v) to c, where n < m
  - c rejects (ignores) the proposal from p'
  - p proposes (m, v') to c, where m > n

- *$P1^a$. An acceptor can accept a proposal numbered n if and only if it has not responded to a prepare request with a number > n*

Example: Livelock



Highest Accept:
(5, “A”)
Highest Prepare:

References

- Marinescu Chapter 2
- Lesie Lamport, *Paxos Made Simple*. ACM SIGACT News, 32(4):51-8, 2001.
  - http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf
- https://angus.nyc/2012/paxos-by-example/
- Seth Gilbert and Nancy Lynch, *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services*. ACM SIGACT News. Volume 33 Issue 2, June 2002, pp 51-59.
  - https://www.comp.nus.edu.sg/~gilbert/pubs/BrewersConjecture-SigAct.pdf
- Eric Brewer, *CAP Twelve years later*. IEEE Computer, Volume:45, Issue: 2
  - http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed