Background on Distributed Computing, Data Management, Peer based Systems

Slides Credit:

Many slides derived from lecture materials (with permission) from UCI's course on Distributed Systems Middleware, UCI's course on transaction processing and distributed databases and Ken Birman's course on Cloud Computing.

.

Cloud Infrastructure

- Large multi-tenant data centers hosting storage, computing, analytics, applications as services.
- Key Public Cloud Players
  - Amazon
  - Salesforce
  - Google
  - Microsoft

What Cloud Computing Provides …

- A storage and data management layer that offers:
  - Extremely low latency of access
  - On demand scalability to very large datasets and users
  - Very high availability
  - Consistency
  - Ability to tolerate failures
- Through the course, we will explore diverse approaches/systems that try to achieve the above.
- These technologies are built upon several decades of work in the fields of storage systems, databases, and distributed computing…

- **Next 2 classes**: existing relevant distributed computing and database technologies.
- **Later**: cloud infrastructure technologies of today.
- **Still later**: key challenges that lie ahead.

Distributed systems 101

Distributed Systems

- Multiple independent computers that appear as one
- Lamport' s Definition
  - " You know you have one when the crash of a computer you have never heard of stops you from getting any work done."

- E.g., Cloud is a highly distributed system with large number of clients, end-users, datacenters, etc.

Classifying Distributed Systems

- Based on degree of synchrony
  - Synchronous
  - Asynchronous
- Based on communication medium
  - Message Passing
  - Shared Memory
- Fault model
  - Crash failures
  - Byzantine failures

Synchrony in distributed systems

- Asynchronous system
  - no assumptions about process execution speeds and message delivery delays

- Synchronous system
  - make assumptions about relative speeds of processes and delays associated with communication channels
  - constrains implementation of processes and communication

Fault Models in Distributed Systems

- Crash failures
  - A processor experiences a crash failure when it ceases to operate at some point without any warning. Failure may not be detectable by other processors.
    - Failstop - processor fails by halting; detectable by other processors.
- Byzantine failures
  - completely unconstrained failures
  - conservative, worst-case assumption for behavior of hardware and software
  - covers the possibility of intelligent (human) intrusion.

Other Fault Models in Distributed Systems

- Dealing with message loss
  - Crash + Link
    - Processor fails by halting. Link fails by losing messages but does not delay, duplicate or corrupt messages.
  - Receive Omission
    - processor receives only a subset of messages sent to it.
  - Send Omission
    - processor fails by transmitting only a subset of the messages it actually attempts to send.
  - General Omission
    - Receive and/or send omission

Replication – a core feature of the Cloud

- To handle more work, make more copies
- If load surges, creating more instances just entails
  - Running more copies on more nodes
  - Adjusting the load-balancer to spray requests to new nodes
- If load drops... just kill the unwanted copies!
  - Little or no warning. Discard any “state” they created locally.
- What do we replicate?
  - Files, storage, data, databases, context, control information
  - Network communication paths (e.g., multiple TCP connections from client to cloud)
  - Computation – for parallelism and /of fault-tolerance

Replicating Data

- Need to differentiate between two types of data
  - "write once" data such as photos that you upload once
  - Data that evolves over time such as current airline tickets available.
- Replication leads to consistency challenges
  - Are the same updates applied to all replicas?
  - Are they applied in the same order?
- Defining consistency requires a notion of time in distributed systems.
- Once we define time, we can be rigorous about notions like "before" or "after" or "simultaneously"
  - conditions for correct replication requires us to to capture such relationships

Time

- In distributed system we need practical ways to deal with time
  - E.g. we may need to agree that update A occurred before update B
  - Or offer a "lease" on a resource that expires at time 10:10.0150
  - Or guarantee that a time critical event will reach all interested parties within 100ms

But what does time "mean"?

- Time on a global clock?
  - E.g. on a city clock tower?
  - ... or perhaps on a GPS receiver?
- … or on a machine's local clock
  - But was it set accurately?
  - And could it drift, e.g. run fast or slow?
  - What about faults, like stuck bits?
- … or could try to agree on time

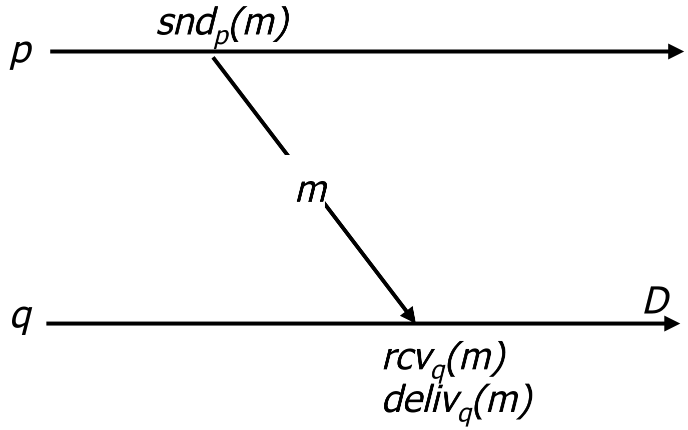
Lamport's approach

- Leslie Lamport suggested that we should reduce time to its basics
  - Time lets a system ask "Which came first: event A or event B?"
  - In effect: time is a means of labeling events so that…
    - If A happened before B, TIME(A) < TIME(B)
    - If TIME(A) < TIME(B), A happened before B

Happens before "relation"

- We say that "A happens before B", written A→B, if
  - A→B according to the local ordering, or
  - A is a send and B is the corresponding receive , or
  - A and B are related under transitive closure of rules (1) and (2)

- Notice that, so far, this is just a mathematical notation, not a "systems tool"
  - Given a trace of what happened in a system we could use these tools to talk about the trace
  - But need a way to "implement" this idea



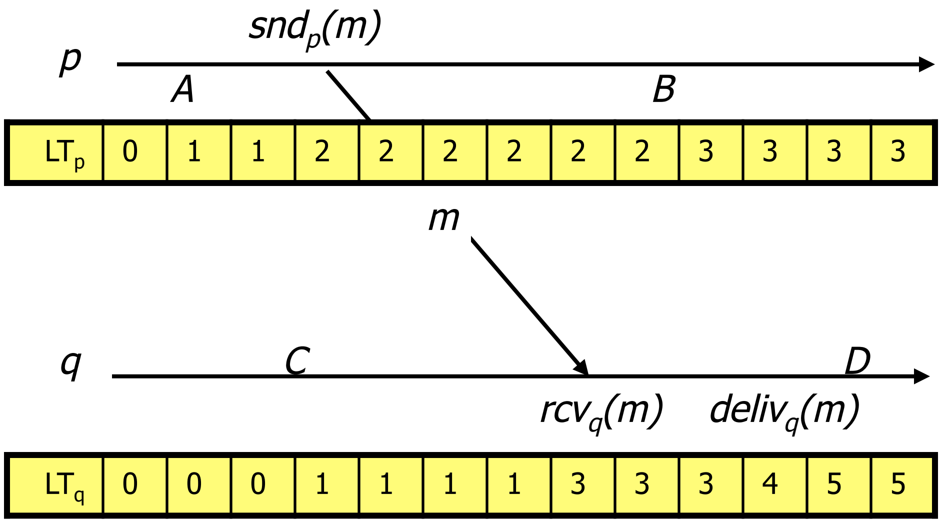
Logical clocks

- A simple tool that can capture parts of the happens before relation
- First version: uses just a single integer
  - Designed for big (64-bit or more) counters
  - Each process p maintains LT_p, a local counter
  - A message m will carry LT_m

Rules for managing logical clocks

- When an event happens at a process p it increments LTp.
  - Any event that matters to p
  - Normally, also snd and rcv events (since we want receive to occur "after" the matching send)
- When p sends m, set
  - LTm = LTp
- When q receives m, set
  - LTq = max(LTq, LTm)+1

Time-line with LT annotations



LT_p	0	1	1	2	2	2	2	2	2	3	3	3	3
LT_q	0	0	0	1	1	1	1	3	3	3	4	5	5

- LT(A) = 1, LT(sndp(m)) = 2, LT(m) = 2
- LT(rcvq(m))=max(1,2)+1=3, etc…

Logical clocks

- If A happens before B, A→B, then LT(A)<LT(B)
- But converse might not be true:
  - If LT(A)<LT(B) can't be sure that A→B
  - This is because processes that don't communicate still assign timestamps and hence events will "seem" to have an order

Can we do better?

- One option is to use vector clocks
- Here we treat timestamps as a list
  - One counter for each process
- Rules for managing vector times differ from what did with logical clocks

History of vector clocks?

- Originated in work at UCLA on file systems that allowed updates from multiple sources concurrently
  - Jerry Popek's FICUS system
  - Today version systems (e.g. SVN, CVS) use the idea

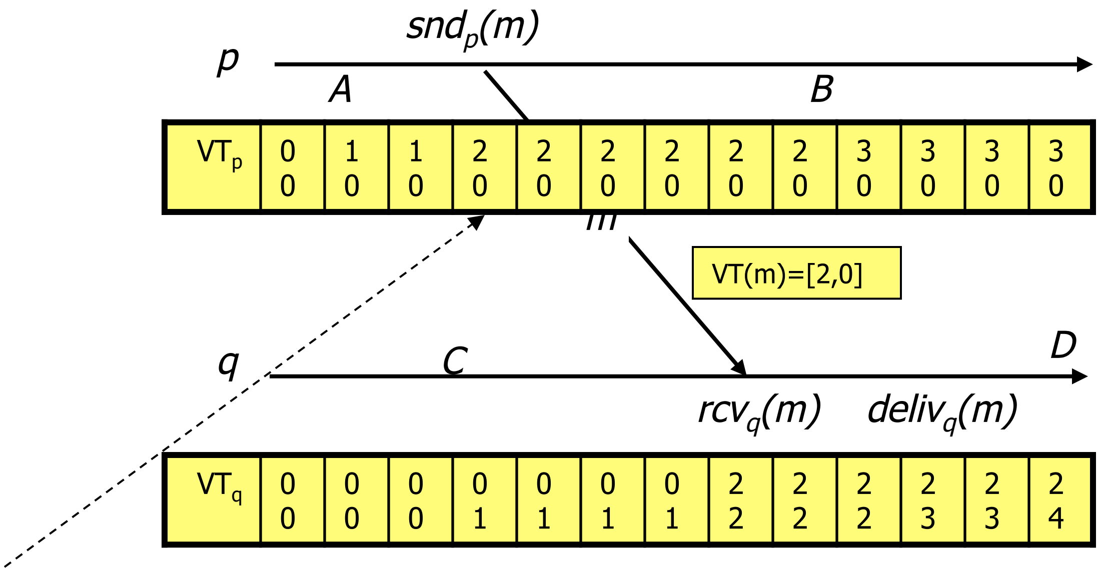
- Also gradually adopted in distributed systems

- Most of the "formal" work was done by Fidge and Mattern in Europe, long after idea was in wide use

Vector clocks

- Clock is a vector: e.g. VT(A)=[1, 0]
  - We'll just assign p index 0 and q index 1
  - Vector clocks require either agreement on the numbering, or that the actual process id's be included with the vector
- Rules for managing vector clock
  - When event happens at p, increment VTp[indexp]
    - Normally, also increment for snd and rcv events
  - When sending a message, set VT(m)=VTp
  - When receiving, set VTq=max(VTq, VT(m))

Time-line with VT annotations



VT_p	0	1	1	2	2	2	2	2	2	3	3	3	3
	0	0	0	0	0	0	0	0	0	0	0	0	0

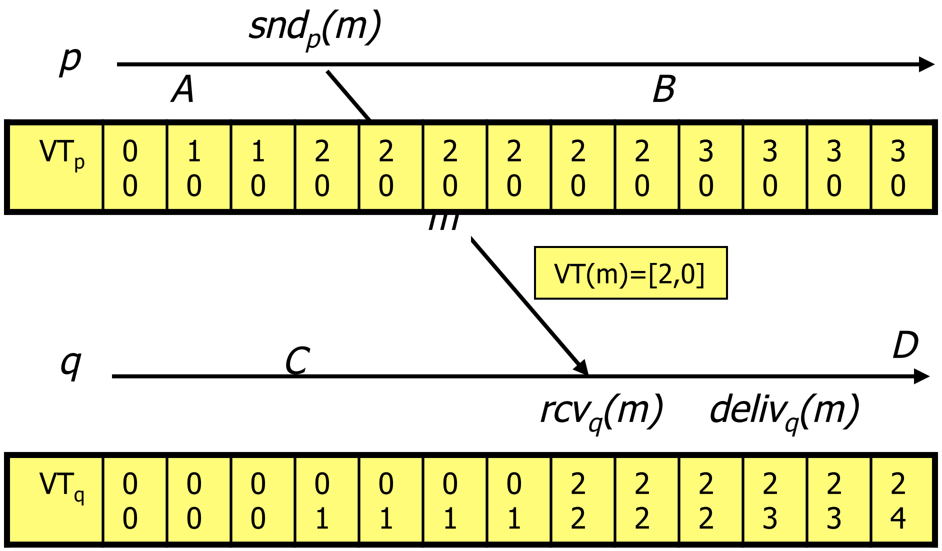
VT_q	0	0	0	0	0	0	0	2	2	2	2	2	2
	0	0	0	1	1	1	1	2	2	2	3	3	4

Could also be [1,0] if we decide not to increment the clock on a snd event. Decision depends on how the timestamps will be used.

Rules for comparison of VTs

- We'll say that VTA ≤ VTB if
  - ∀I, VTA[i] ≤ VTB[i]
- And we'll say that VTA < VTB if
  - VTA ≤ VTB but VTA ≠ VTB
  - That is, for some i, VTA[i] < VTB[i]
- Examples?
  - [2,4] ≤ [2,4]
  - [1,3] < [7,3]
  - [1,3] is "incomparable" to [3,1]

Time-line with VT annotations



VT_p	0	1	1	2	2	2	2	2	2	3	3	3	3
	0	0	0	0	0	0	0	0	0	0	0	0	0

VT_q	0	0	0	0	0	0	0	2	2	2	2	2	2
	0	0	0	1	1	1	1	2	2	2	3	3	4

- VT(A)=[1,0]. VT(D)=[2,4]. So VT(A)<VT(D)
- VT(B)=[3,0]. So VT(B) and VT(D) are incomparable

Vector time and happens before

- If A→B, then VT(A)<VT(B)
  - – Write a chain of events from A to B
  - – Step by step the vector clocks get larger
- If VT(A)<VT(B) then A→B
  - – Two cases: if A and B both happen at same process p, trivial
  - – If A happens at p and B at q, can trace the path back by which q “learned” VTA[p]
- Otherwise A and B happened concurrently

Mutual Exclusion

- Often, to implement consistency in replicated resources, we require processes to have exclusive access to shared resources.
  - General concept in OS, specialized to locking in databases
- Many protocols both for centralized and distributed systems
- Simple centralized solution:
  - One process becomes the coordinator. Maintains request queue. Grants request one at a time.
  - Simple to generalize to a fully distributed approach.
    - will suffer from resource availability in presence of failures.
    - Quorums introduced to improve availability in presence of failures

Quorum-Based Protocol

- Define conflicting operations over resource
  - Typically, read() and write()
  - read() and write() and two write() conflict, but two read() do not.
- Quorum = set of processes from which to seek permission for an operation.
  - Read_Quorum(X) = the set of processes to request permission to read X
  - Write_Quorum(X) = set of processes to request permission to write X
- Read and write quorums and two write quorums for a resource must overlap!
- Lots of quorum based protocols can be designed.

Simple Quorum-Based Protocol

- Data item with 6 replicas
- Read_Quorum = any subset of 3 replicas
- Write_Quorum = any subset of 4 replicas
- To read, a process must seek permission from one read quorum. Likewise to update, a process must find a write quorum
- Note that two updates /update and a read cannot execute concurrently.
- With quorums, the system remains available even in presence of failures (unless failures prevent reaching a quorum).
- Many generalizations to the above protocol including exploration of overhead/availability tradeoffs for different workloads – e.g., read-heavy/update-heavy/balanced workloads

Leadership Election

- Many distributed algorithms require one process to be selected as leader / coordinator.
  - It does not matter which process is elected.
  - What is important is that one and only one process is chosen and all processes agree on this decision.

- Assume that each process has a unique number (identifier).
  - In general, election algorithms attempt to locate the process with the highest number, among those which currently are up.

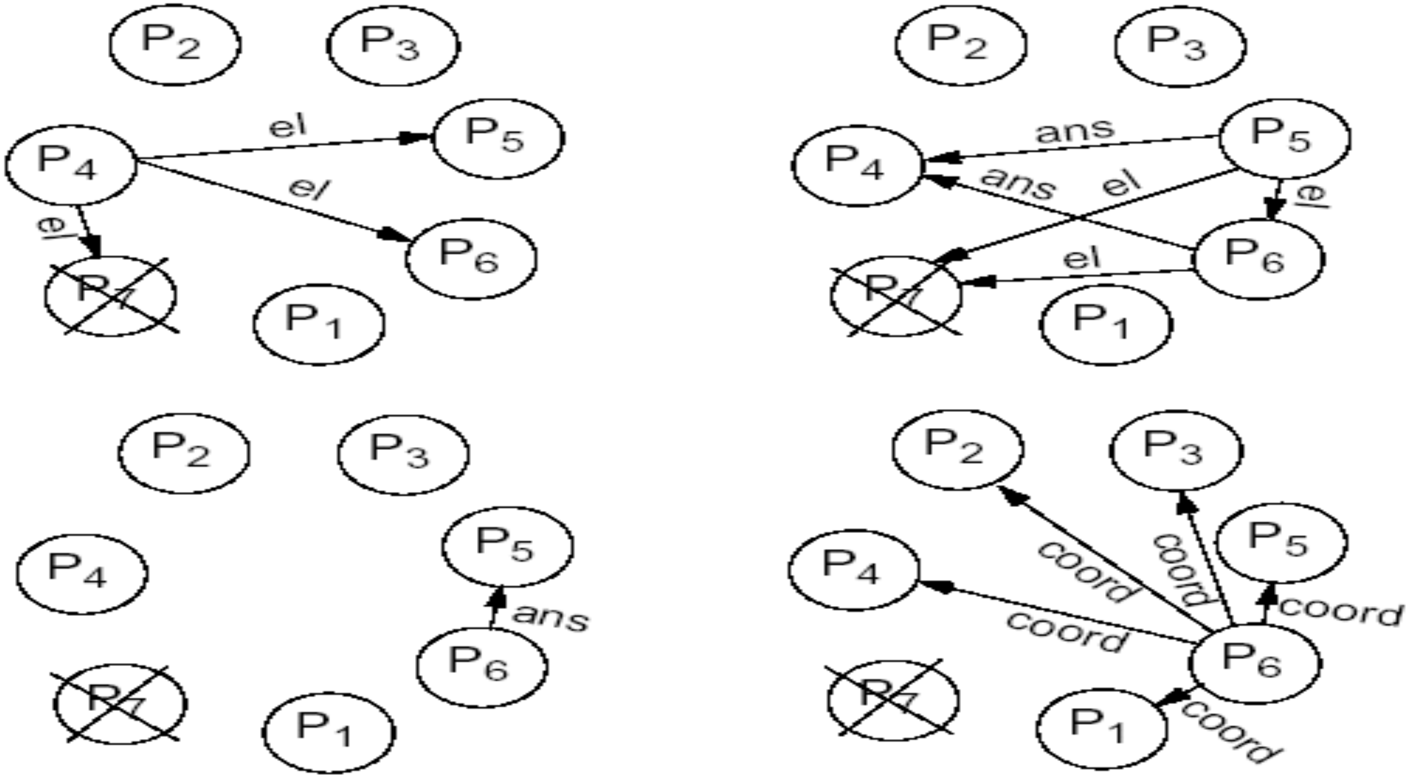
- Election is typically started after a failure occurs.
  - The detection of a failure (e.g. the crash of the current coordinator) is normally based on time-out → a process that gets no response for a period of time suspects a failure and initiates an election process.

- An election process is typically performed in two phases:
  - Select a leader with the highest priority.
  - Inform all processes about the winner.

The Bully Algorithm

- A process has to know the identifier of all other processes
  - (it doesn't know, however, which one is still up); the process with the highest identifier, among those which are up, is selected.
- Any process could fail during the election procedure.
- When a process Pi detects a failure and a coordinator has to be elected
  - it sends an election message to all the processes with a higher identifier and then waits for an answer message:
  - If no response arrives within a time limit
    - Pi becomes the coordinator (all processes with higher identifier are down)
    - it broadcasts a coordinator message to all processes to let them know.
  - If an answer message arrives,
    - Pi knows that another process has to become the coordinator → it waits in order to receive the coordinator message.
    - If this message fails to arrive within a time limit (which means that a potential coordinator crashed after sending the answer message) Pi resends the election message.
- When receiving an election message from Pi
  - a process Pj replies with an answer message to Pi and
  - then starts an election procedure itself(unless it has already started one) it sends an election message to all processes with higher identifier.
- Finally all processes get an answer message, except the one which becomes the coordinator.

The Bully Algorithm (cont'd)



- If P_6 crashes before sending the coordinator message, P_4 and P_5 restart the election process.

☞ The best case: the process with the second highest identifier notices the coordinator's failure. It can immediately select itself and then send n-2 coordinator messages.

☞ The worst case: the process with the lowest identifier initiates the election; it sends n-1 election messages to processes which themselves initiate each one an election ⇒ $O(n^2)$ messages.

Group Communication

- Often, processes need to send a message to a set of processes
  - Broadcast -- send message to all the sites in the system
  - Multicast – send message to a subset of sites in the system
- Group communication often used to propagate updates to replicas.

Group Communication Order Guarantees

- FIFO – messages delivered in order they are sent (by any single sender)
  - Easy to implement by ordering messages. Message delivery waits until all previous messages delivered
- Causal Order – if message m1 “happened before” m2, then m1 delivered before m2 on every common destination.
  - Can be implemented by requiring message to carry all causally preceding messages – Expensive!
  - Use vector clocks to delay delivery of message until all causally preceding messages delivered
- Total Order (atomic) – messages are delivered in the same order at all destinations.
  - Distributed multi-round protocol needed to agree upon the delivery order

Group Communication Membership

- Group communication becomes more challenging when new members can join (or leave) a group.

- Members must have a consistent view of the group.

Group Communication with Dynamic Membership

- Atomic Multicast
  - Message is delivered to all processes or to none at all. May also require that messages are delivered in the same order to all processes.
- Failure Atomicity
  - Failures do not result in incomplete delivery of multicast messages or holes in the causal delivery order
- Uniformity
  - A view change reported to a member is reported to all other members
- Liveness
  - A machine that does not respond to messages sent to it is removed from the local view of the sender within a finite amount of time.

Consensus

- N processes
- Each process p has
  - input variable xp : initially either 0 or 1
  - output variable yp : initially b (b=undecided) – can be changed only once
- Consensus problem: design a protocol so that either
  - all non-faulty processes set their output variables to 0
  - Or all non-faulty processes set their output variables to 1
  - There is at least one initial state that leads to each outcomes 1 and 2 above
- Solutions to consensus problem can be used to implement agreement about events, event ordering, and hence replication

Solving Consensus

- If no failures – trivial
  - Just use a broadcast from all to all.
- If failures
  - Depends if the system is synchronous or asynchronous

Solving Consensus: Synchronous System

- [Dolev 82] No solution if fewer than 3f+1 processes, if f of them can fail maliciously (byzantine failure).

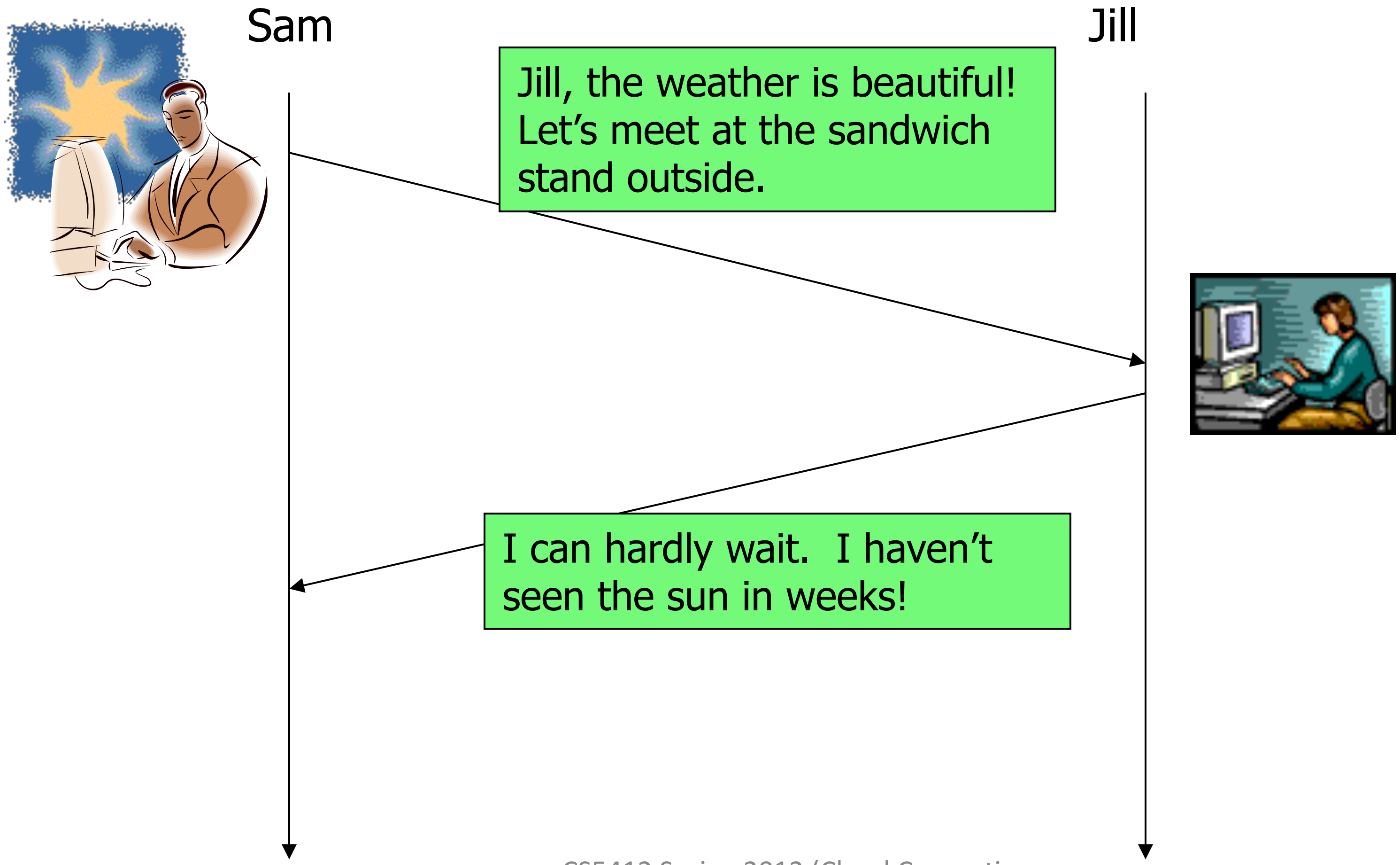
Solving Consensus in Asynchronous Systems

- Surprising result by Fischer, Lynch, and Patterson:
  - Impossibility of Asynchronous Distributed Consensus with a Single Faulty Process
- They prove that no asynchronous algorithm for agreeing on a one-bit value can guarantee that it will terminate in the presence of crash faults
  - And this is true even if no crash actually occurs!
  - Proof constructs infinite non-terminating runs

Intuition Behind FLP Impossibility Theorem

- Jill and Sam will meet for lunch. They'll eat in the cafeteria unless both are sure that the weather is good
  - Jill's cubicle is inside, so Sam will send email
  - Both have lots of meetings, and might not read email. So she'll acknowledge his message.
  - They'll meet inside if one or the other is away from their desk and misses the email.
- Sam sees sun. Sends email. Jill acks's. Can they meet outside?

Sam and Jill



Sam

Jill

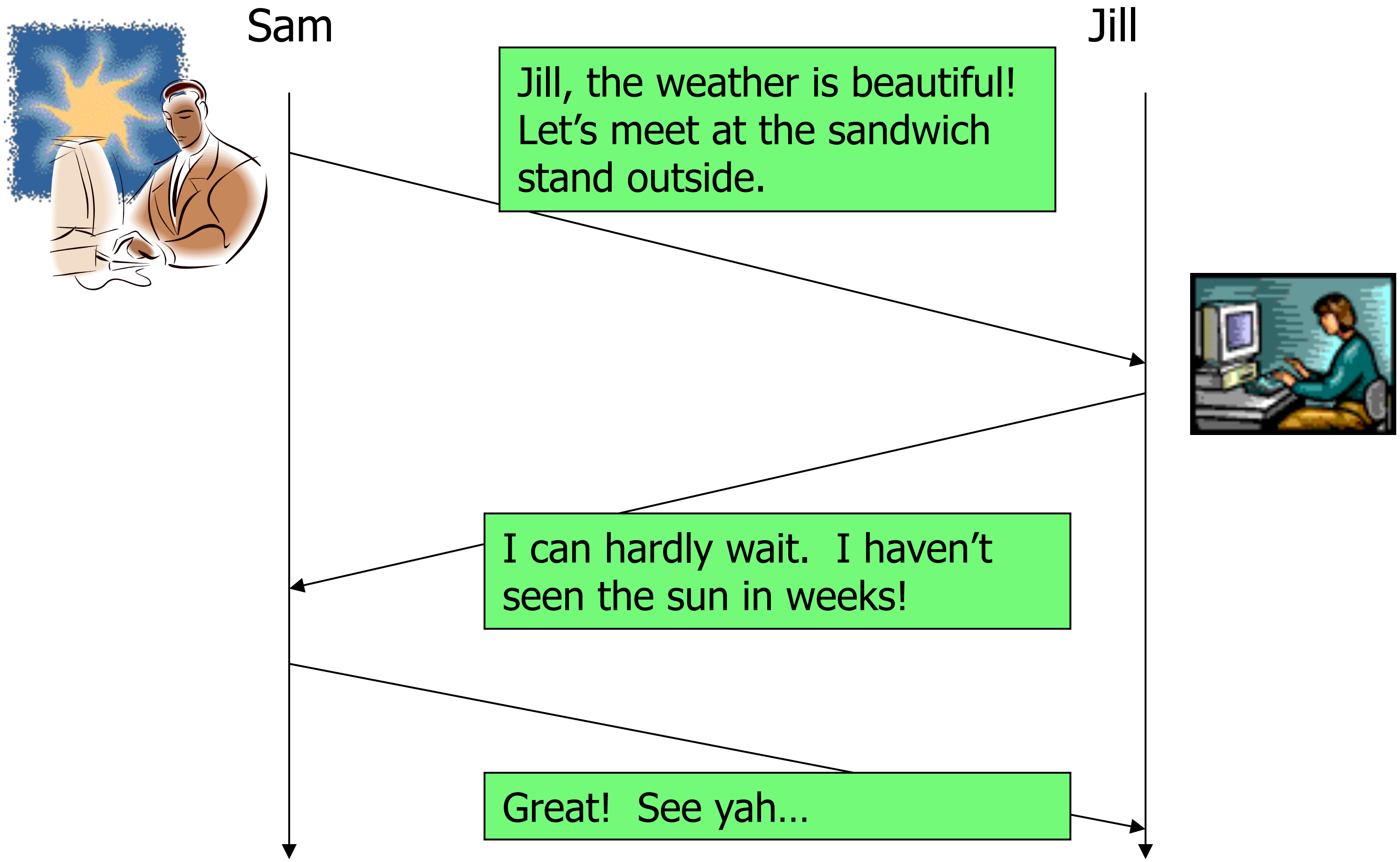
Jill, the weather is beautiful! Let’s meet at the sandwich stand outside.

I can hardly wait. I haven’t seen the sun in weeks!

They eat inside! Sam reasons:

- “Jill sent an acknowledgement but doesn’t know if I read it
- “If I didn’t get her acknowledgement I’ll assume she didn’t get my email
- “In that case I’ll go to the cafeteria
- “She’s uncertain, so she’ll meet me there

Sam had better send an Ack



Jill, the weather is beautiful! Let’s meet at the sandwich stand outside.

I can hardly wait. I haven’t seen the sun in weeks!

Great! See yah…

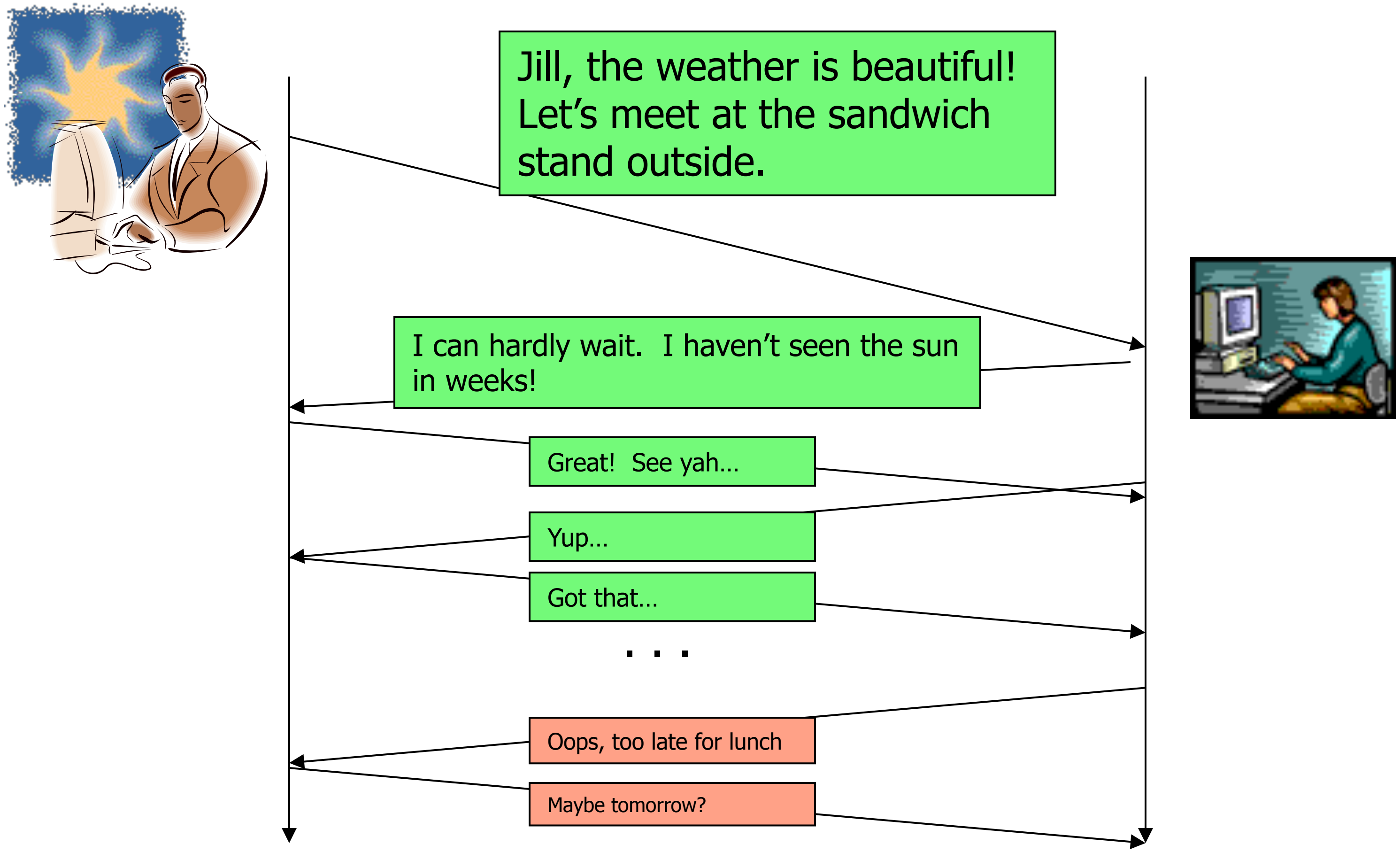
Why didn’t this help?

- Jill got the ack… but she realizes that Sam won’t be sure she got it
- Being unsure, he’s in the same state as before
- So he’ll go to the cafeteria, being dull and logical. And so she meets him there.

New and improved protocol

- Jill sends an ack. Sam acks the ack. Jill acks the ack of the ack….
- Suppose that noon arrives and Jill has sent her 117’th ack.
  - Should she assume that lunch is outside in the sun, or inside in the cafeteria?

How Sam and Jill’s romance ended



Jill, the weather is beautiful! Let’s meet at the sandwich stand outside.

I can hardly wait. I haven’t seen the sun in weeks!

Great! See yah…

Yup…

Got that…

. . .

Oops, too late for lunch

Maybe tomorrow?

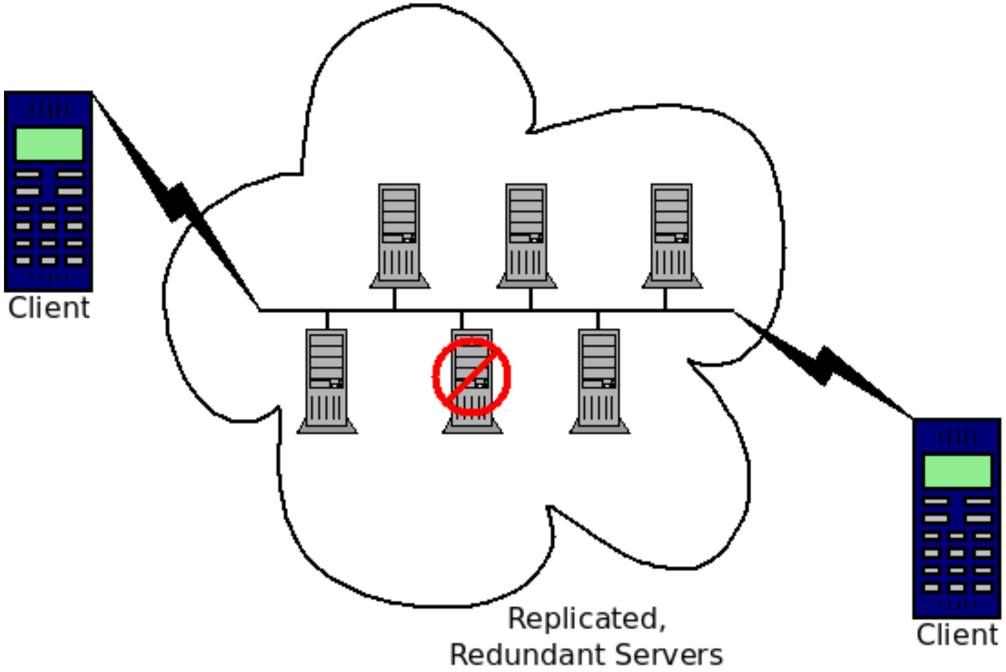
Things we just can’t do

- We can’t detect failures in a trustworthy, consistent manner
- We can’t reach a state of “common knowledge” concerning something not agreed upon in the first place
- We can’t guarantee agreement on things (election of a leader, update to a replicated variable) in a way certain to tolerate failures

But what does it mean?

- In formal proofs, an algorithm is totally correct if
  - It computes the right thing
  - And it always terminates
- When we say something is possible, we mean "there is a totally correct algorithm" solving the problem
- FLP proves that any fault-tolerant algorithm solving consensus has runs that never terminate
  - These runs are extremely unlikely ("probability zero")
  - Yet they imply that we can't find a totally correct solution
  - And so "consensus is impossible" ("not always possible")
- In practice, fault-tolerant consensus is ..
  - Definitely possible.
  - E.g. **Paxos [Lamport 1998, 2001] that has become quite popular – discussed next!**

Distributed State Machine



- Fault-tolerance through replication.
  - Need to ensure that replicas remain consistent.
  - Replicas must process requests in the same order.

Consensus Problem (revisit)

- N processes
- Each process p has
  - input variable xp : initially either 0 or 1
  - output variable yp : initially b (b=undecided) – can be changed only once
- Consensus problem: design a protocol so that either
  - all non-faulty processes set their output variables to 0
  - Or all non-faulty processes set their output variables to 1
  - There is at least one initial state that leads to each outcomes 1 and 2 above

The Distributed Consensus Problem

- Simple solution:
  - A single node acts as the “decider.”
    - But this is not fault tolerant. (What if the decider fails?)
- A better solution: Paxos

Fault-Tolerant Consensus

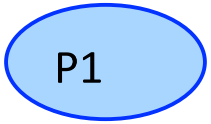
- Requirements
  - Safety
    - Only a value that has been proposed may be chosen.
    - Only a single value is chosen.
    - A process never learns that a value has been chosen until it actually has been.
- Goals
  - Liveness
    - FLP Impossibility Proof (1985)

Assumptions

- Failures
  - “Fail Stop” assumption
    - When a node fails, it ceases to function entirely.
    - May resume normal operation when restarted.
  - Messages
    - May be lost.
    - May be duplicated.
    - May be delayed (and thus reordered).
    - May ***not*** be corrupt.
- Stable Storage

Paxos Terms

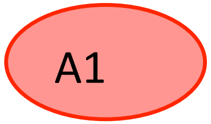
- ***Proposer***



P1

  - Suggests values for consideration by Acceptors.
  - Advocates for a client.

- ***Acceptor***



A1

  - Considers the values proposed by proposers.
  - Renders an accept/reject decision.

- ***Learner***
  - Learns the chosen value.

- In practice, each node will usually play all three roles.

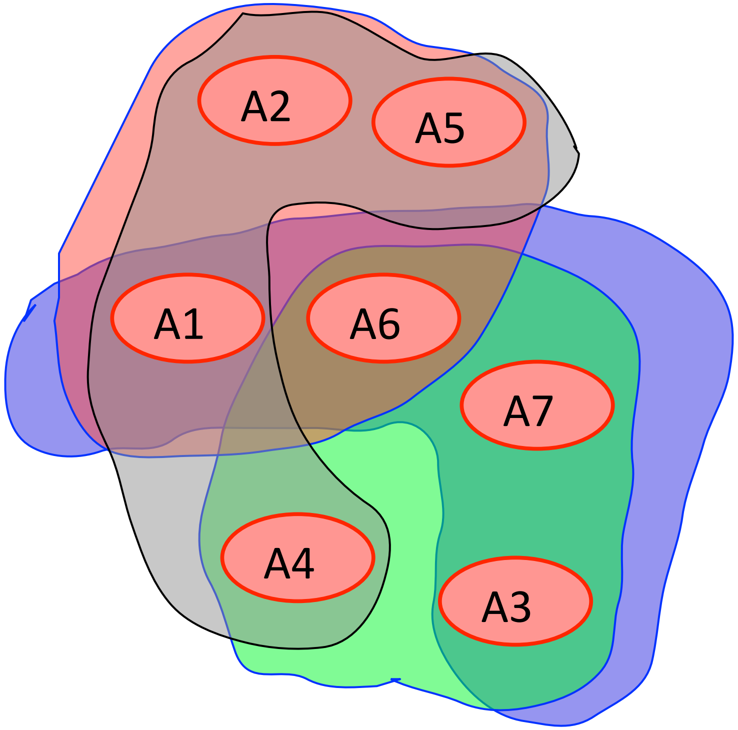
- ***Proposal***
  - An alternative proposed by a proposer.
  - Consists of a unique *number* and a proposed *value*.

    (42, B)

- **We say a value is *chosen* when consensus is reached on that value.**

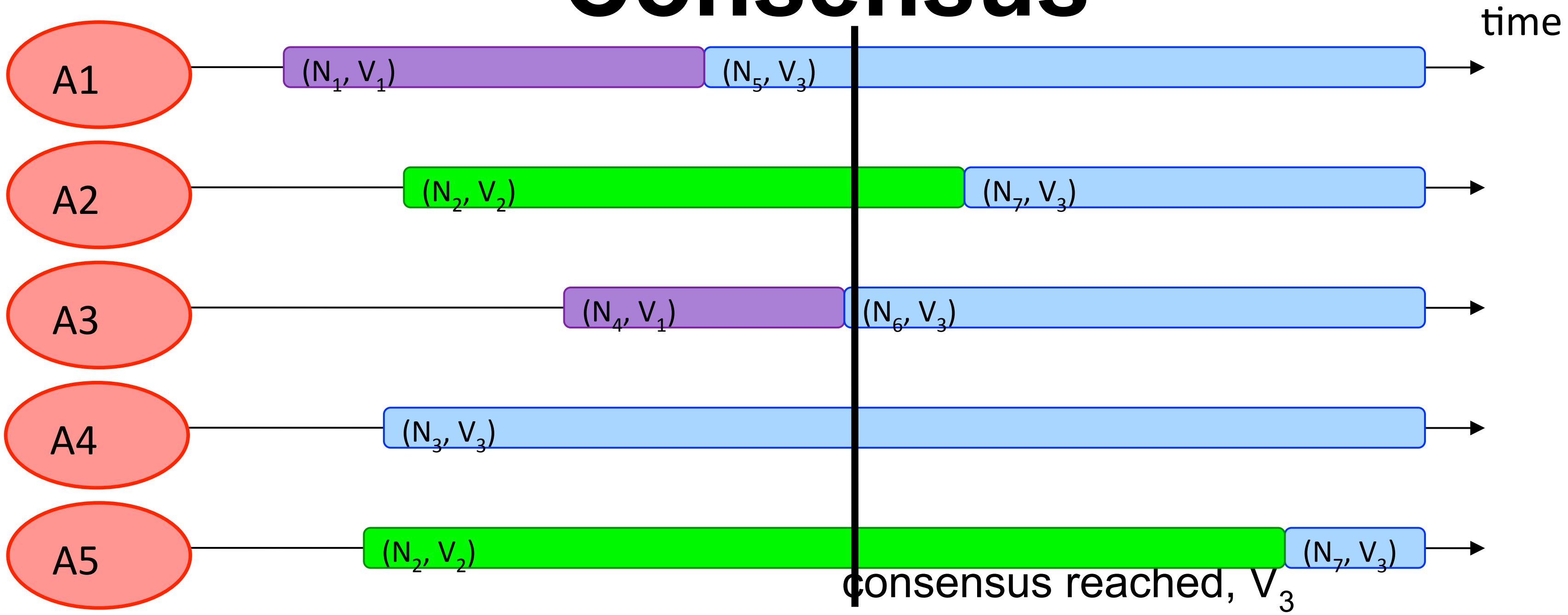
Strong Majority

- "Strong Majority" / "Quorum"
  - A set of acceptors consisting of more than half of all acceptors.
- Any two quorums have a nonempty intersection.
- Helps avoid "split-brain" problem.
  - Acceptors decisions are not in agreement.
  - Common node acts as "tie-breaker."
- In a system with 2F+1 acceptors, F acceptors can fail and we'll be OK.



Quorums in a system with seven acceptors.

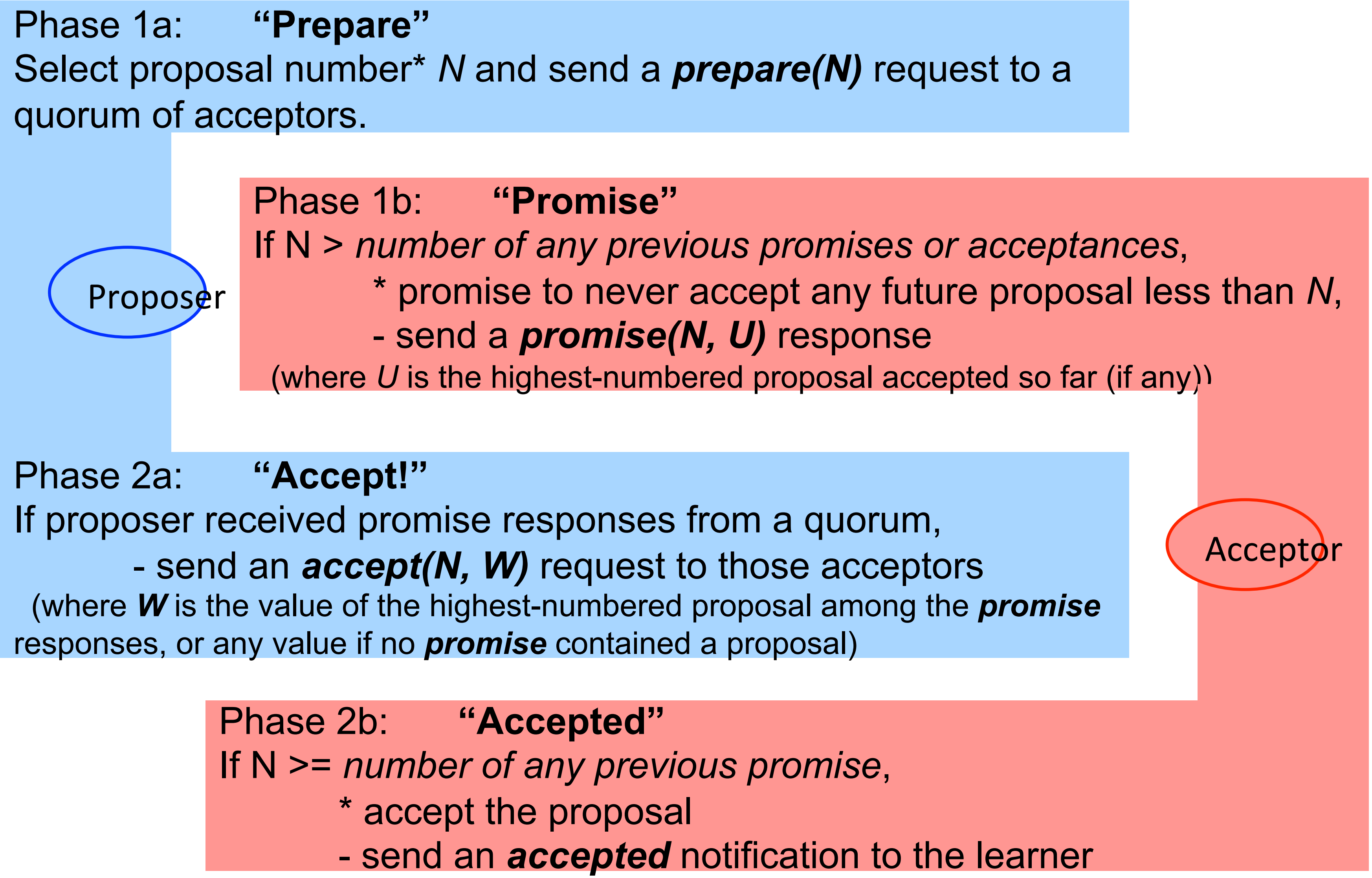
Consensus



consensus reached, V_3 chosen

- Values proposed by proposers are constrained so that once consensus has been reached, all future proposals will carry the chosen value.
- 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.

Basic Paxos Algorithm



Phase 1a: **“Prepare”**

Select proposal number* *N* and send a ***prepare(N)*** request to a quorum of acceptors.

Proposer

Phase 1b: **“Promise”**

If N > *number of any previous promises or acceptances*,

* promise to never accept any future proposal less than *N*,

- send a ***promise(N, U)*** response

(where *U* is the highest-numbered proposal accepted so far (if any))

Phase 2a: **“Accept!”**

If proposer received promise responses from a quorum,

- send an ***accept(N, W)*** request to those acceptors

(where ***W*** is the value of the highest-numbered proposal among the ***promise*** responses, or any value if no ***promise*** contained a proposal)

Acceptor

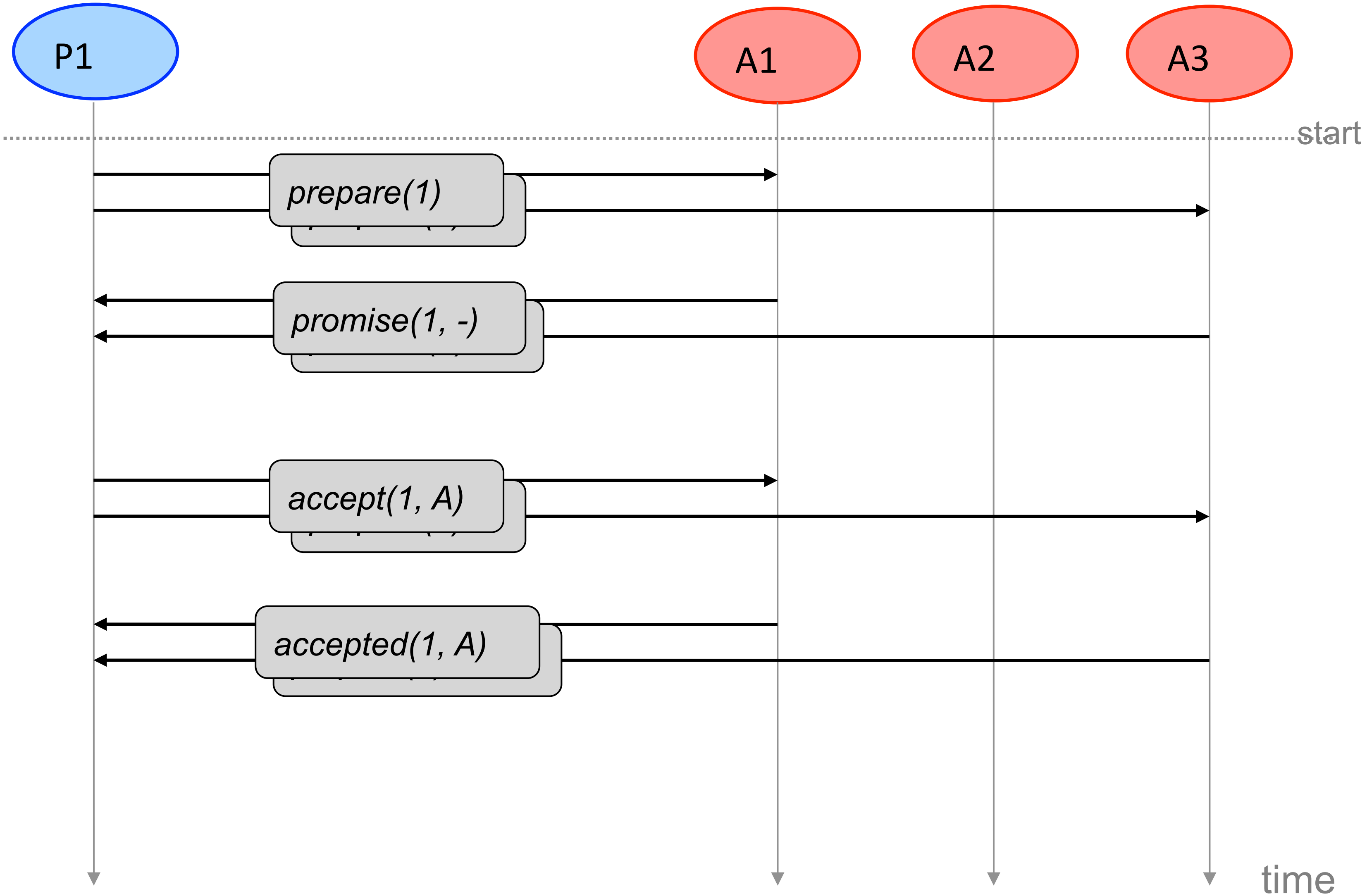
Phase 2b: **“Accepted”**

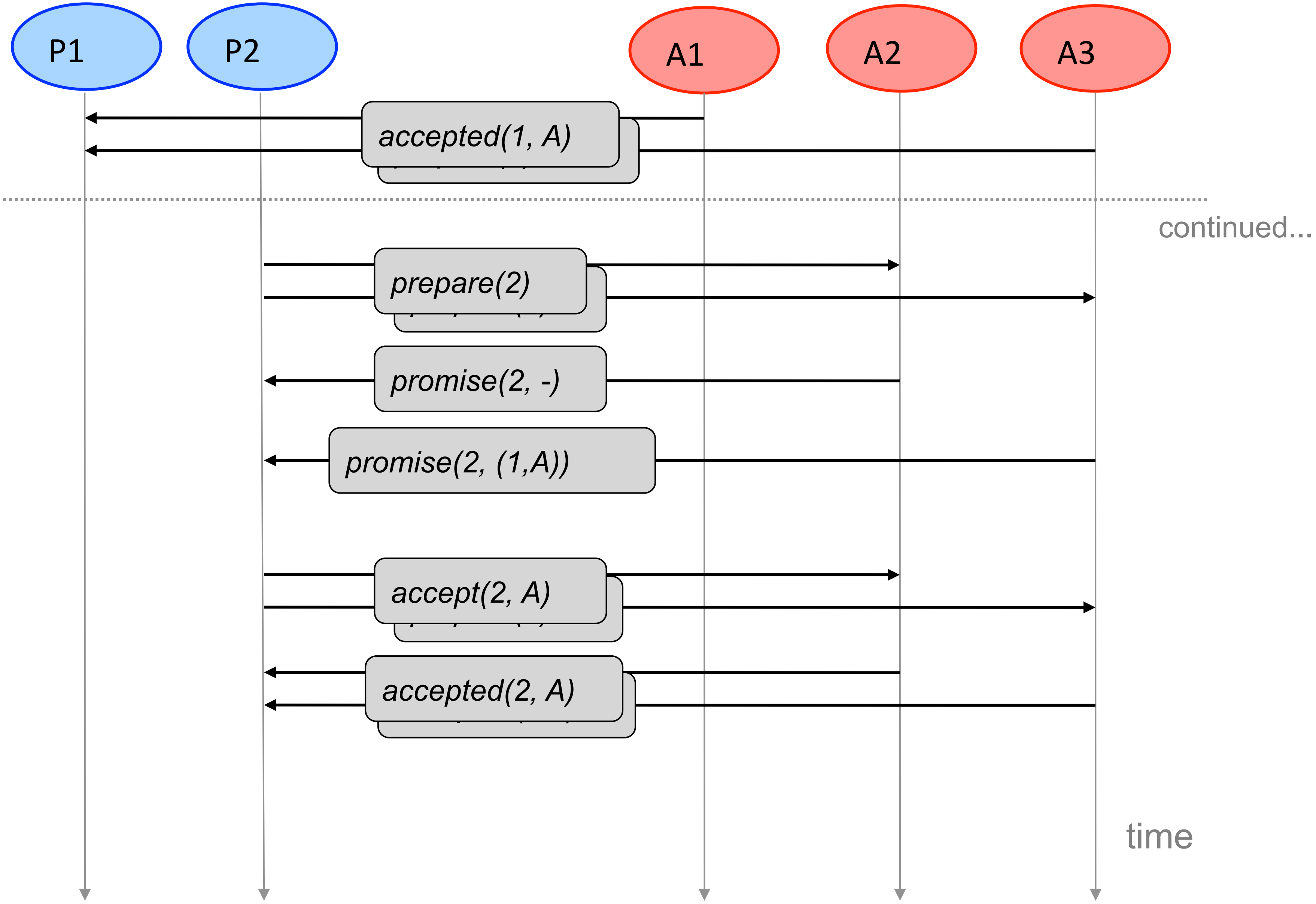
If N >= *number of any previous promise*,

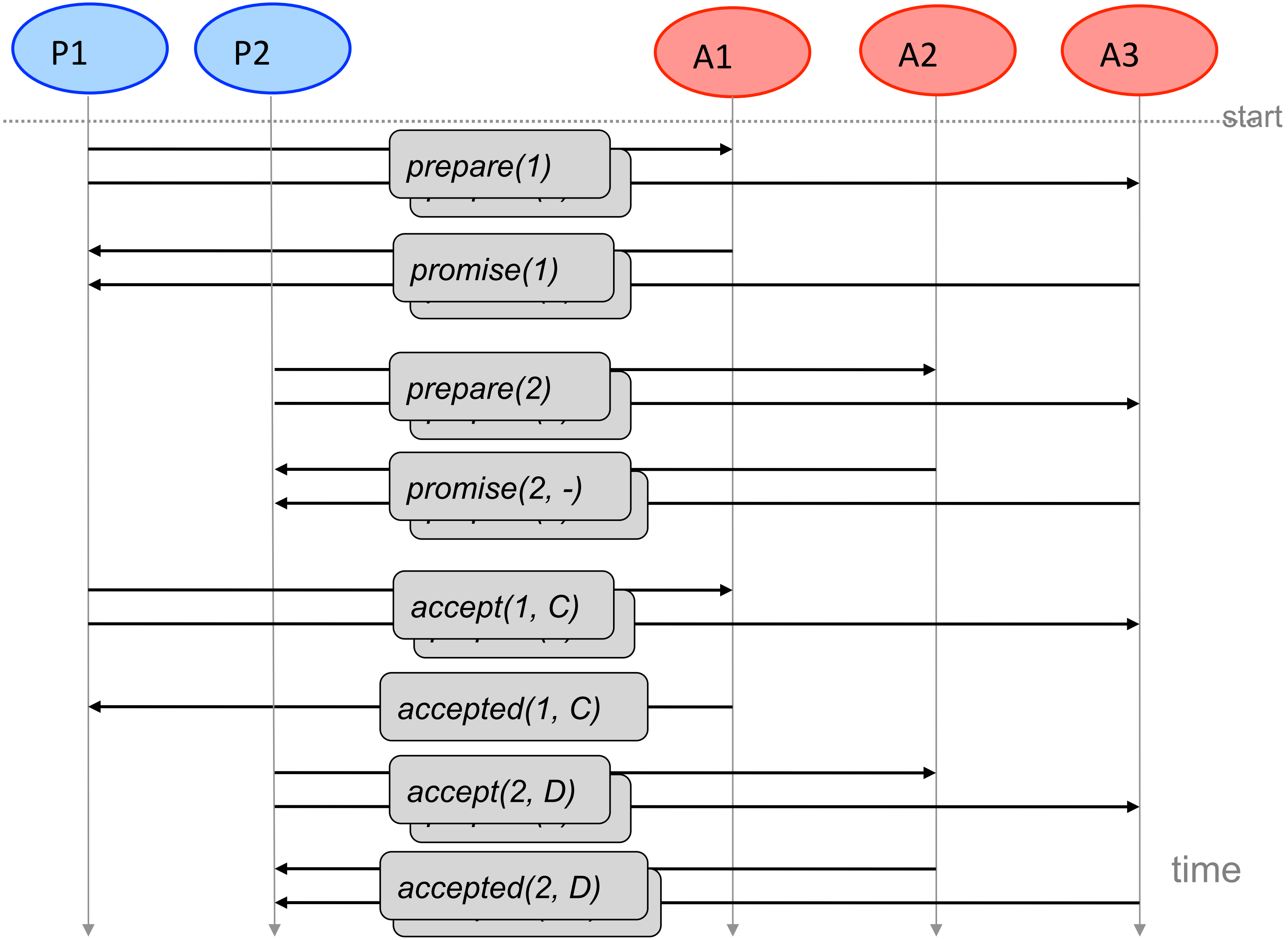
* accept the proposal

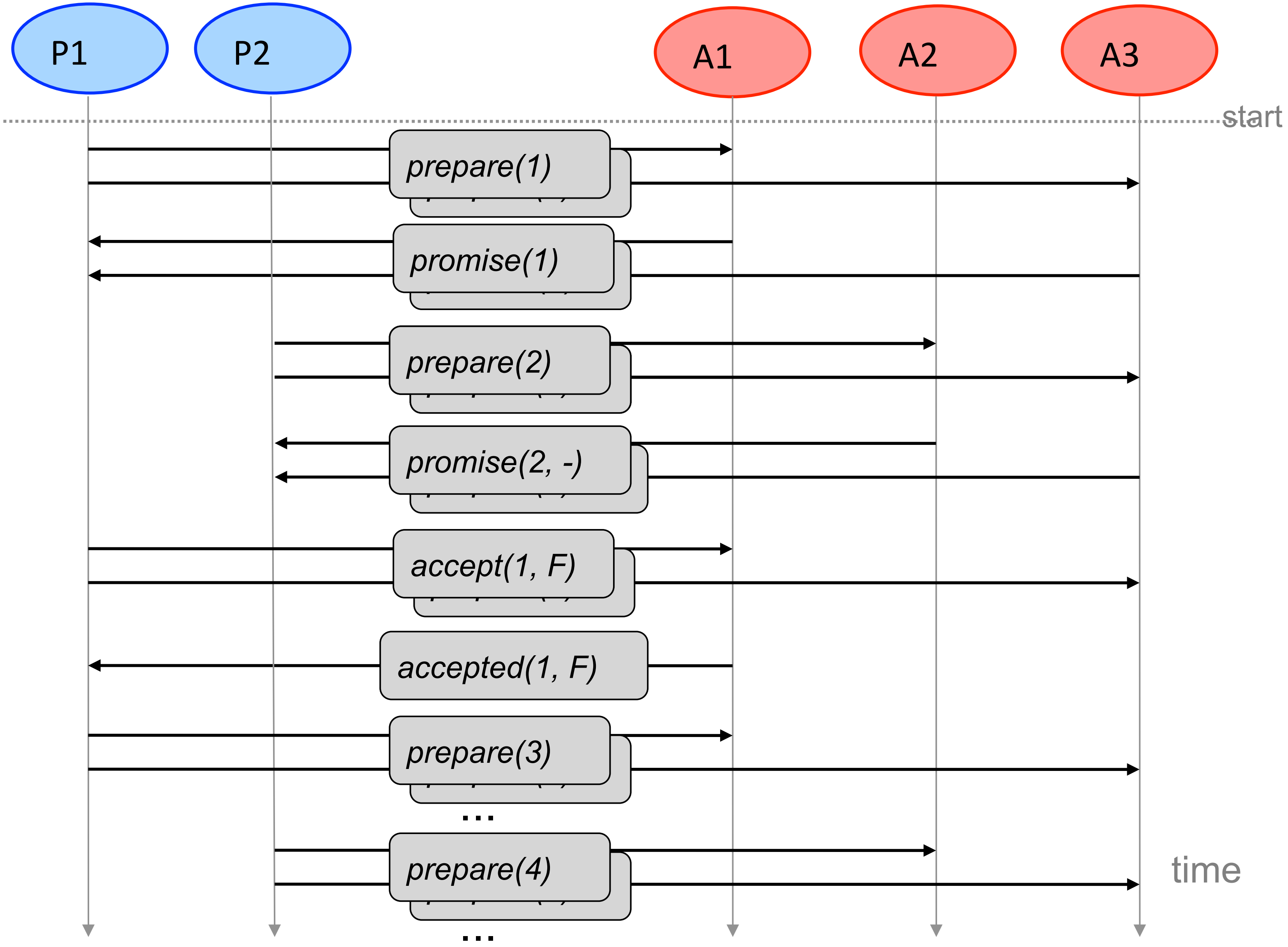
- send an ***accepted*** notification to the learner

* = record to stable storage


P1
A1
A2
A3
start
prepare(1)
promise(1, -)
accept(1, A)
accepted(1, A)
time


P1
P2
A1
A2
A3
accepted(1, A)
continued...
prepare(2)
promise(2, -)
promise(2, (1,A))
accept(2, A)
accepted(2, A)
time


P1
P2
A1
A2
A3
start
prepare(1)
promise(1)
prepare(2)
promise(2, -)
accept(1, C)
accepted(1, C)
accept(2, D)
accepted(2, D)
time


P1
P2
A1
A2
A3
start
prepare(1)
promise(1)
prepare(2)
promise(2, -)
accept(1, F)
accepted(1, F)
prepare(3)
…
prepare(4)
time
…

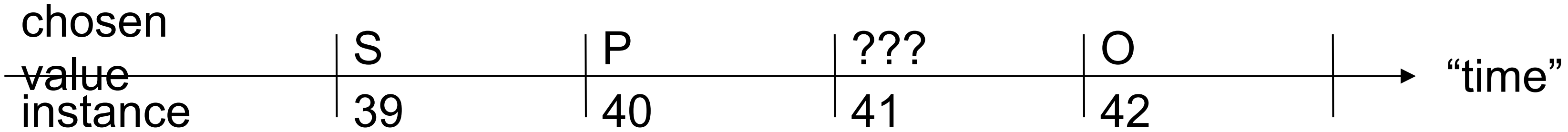
Other Considerations

- Liveness
  - Can't be guaranteed in general.
  - Distinguished Proposer
    - All proposals are funneled through one node.
  - Can re-elect on failure.

- Learning the Chosen Value
  - Acceptors notify some set of learners upon acceptance.
  - Distinguished Learner

- A node may play the role of both distinguished proposer and distinguished learner – we call such a node the ***master***.

Multi-Paxos



- A single ***instance*** of Paxos yields a single chosen value.
- To form a sequence of chosen values, simply apply Paxos iteratively.
    - To distinguish, include an instance number in messages.

- Facilitates replication of a state machine.

Paxos Variations

- Cheap Paxos
  - Reconfiguration
    - Eject failed acceptors.
    - Fault-tolerant with only F+1 nodes (vs 2F+1).
    - Failures must not happen too quickly.

- Fast Paxos
  - Clients send *accept* messages to acceptors.
  - Master is responsible for breaking ties.
  - Reduces message traffic.

- Byzantine Paxos
  - Arbitrary failures – lying, collusion, fabricated messages, selective non-participation.
  - Adds an extra “verify” phase to the algorithm.

Summary..

- We have learnt some of the basics of distributed computing
  - Logical time, vector clock, happened before relationships, quorums, messaging, multicast, broadcast, impossibility of consensus, practical consensus protocols – e.g., Paxos
- We motivated all this from the need to implement replication (i.e., making copies of data, files, objects)
- We replicate data (i.e., make copies) for various purposes
  - Scaling the application (allowing replica to be processed at different machines, keeping latency lower (keeping data closer to where it is needed thereby avoiding internet latencies, etc.), fault-tolerance (data available even when some sites fail).
- But replication raises the consistency challenge.

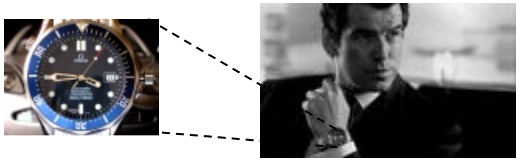
What is Consistency?

- We would say that a replicated entity behave in a consistent manner if it mimics the behavior of a non-replicated entity
  - E.g. if I ask it some question, and it answers, and then you ask it that question, your answer is either the same or reflects some update to the underlying state
  - Many copies but acts like just one

- An inconsistent service is one that seems "broken"

Consistency lets us ignore implementation

A consistent distributed system will often have many components, but users observe behavior indistinguishable from that of a single-component reference system



Reference Model

Implementation

But Can We Implement Consistency in Clouds?

- **Eric Brewer's CAP theorem**: (2000 PODC Keynote) We can only have 2 of the three properties -- Consistency, Availability, and Network Partition tolerance
- A proof of CAP was later introduced by MIT's Seth Gilbert and Nancy Lynch
  - Suppose a data center service is active in two parts of the country with a wide-area Internet link between them
  - We temporarily cut the link ("partitioning" the network)
  - And present the service with conflicting requests
- The replicas can't talk to each other so can't sense the conflict
- If they respond at this point, inconsistency arises

Interplay with CAP

- In Large scale operations – be prepared for network partitions.
- So effectively, we have to choose between consistency or availability.
- RDBMS choose consistency (as we will see)
- In cloud applications:
  - response has to be very fast, hence availability cannot be sacrificed.
  - Furthermore, data centers should be responsive even if a transient fault makes it hard to reach some service. So they should use cached data to respond faster even if the cached entry can't be validated and might be stale
- Many cloud solutions thus choose weaker consistency for faster response.

C and A: In a Network Partition

- Dynamo – quorum based replication
  - Multi-mastering keys – Eventual Consistency
  - Tunable read and write quorums
  - Larger quorums – higher consistency, lower availability
  - Vector clocks to allow application supported reconciliation
- PNUTS – log based replication
  - Similar to log replay – reliable log multicast
  - Per record mastering – timeline consistency
  - Major outage might result in losing the tail of the log

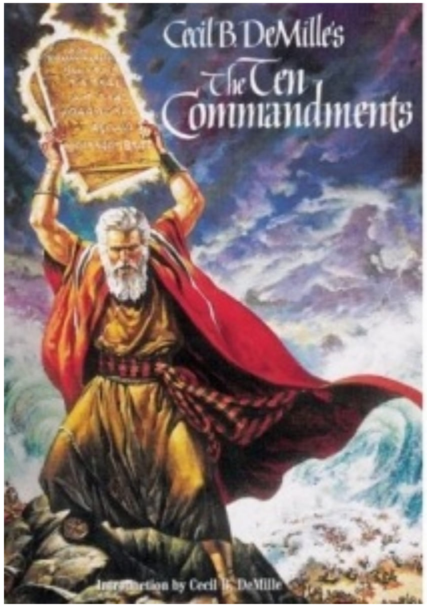
Is inconsistency a bad thing?

- How much consistency is really needed by applications in the cloud?
  - Think about YouTube videos. Would consistency be an issue here?
  - What about the Amazon “number of units available” counters. Will people notice if those are a bit off?
  - Even classic database technology ships by default in lower isolation levels. Long history of research on relaxing consistency requirements (e.g., compensation, heuristic commit, multidatabases, etc.)
- Puzzle: is there a theoretical framework to establish how much consistency a given thing needs?

The Wisdom of the Sages

eBay’s Five Commandments

- As described by Randy Shoup at LADIS 2008
- Thou shalt…
  - – 1. Partition Everything
  - – 2. Use Asynchrony Everywhere
  - – 3. Automate Everything
  - – 4. Remember: Everything Fails
  - – 5. Embrace Inconsistency



Vogels at the Helm

- Werner Vogels is CTO at Amazon.com…
- He was involved in building a new shopping cart service
  - The old one used strong consistency for replicated data
  - New version was build over a DHT, like Chord, and has weak consistency with eventual convergence

- This weakens guarantees… but
  - Speed matters more than correctness

Consistency

Consistency technologies just don’t scale!

But Inconsistency is a "big headache" for Application Writers

"I love **eventual consistency** but there are some applications that are much easier to implement with strong consistency. Many like eventual consistency because it allows us to scale-out nearly without bound ***but it does come with a cost in programming model complexity***."

February 24, 2010

James Hamilton's Blog

Sacrificing Consistency

Increased Application Complexity

```
public void confirm_friend_request(user1, user2)
{
begin_transaction();

        update_friend_list(user1, user2, status.confirmed);
         //Palo Alto
        update_friend_list(user2, user1, status.confirmed);
        //London

end_transaction();
}
```

```
public void confirm_friend_request_A(user1, user2){
  try{
      update_friend_list(user1, user2, status.confirmed); //palo alto
   }
   catch(exception e){
      report_error(e);  return;
  }
 try{
     update_friend_list(user2, user1, status.confirmed); //london
 }
 catch(exception e) {
    revert_friend_list(user1, user2);
    report_error(e);
    return;
  }
}
```

```
public void confirm_friend_request_B(user1, user2){
try{
   update_friend_list(user1, user2, status.confirmed); //palo alto
 }catch(exception e){
  report_error(e);
  add_to_retry_queue(operation.updatefriendlist, user1, user2,
current_time());
 }
 try{
  update_friend_list(user2, user1, status.confirmed); //london
 }catch(exception e) {
  report_error(e);
  add_to_retry_queue(operation.updatefriendlist, user2, user1,
current_time());
 }
}
```

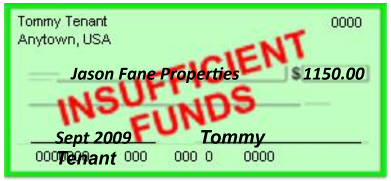
It gets too complicated with Eventual Consistency

```
/* get_friends() method has to reconcile results returned by get ecause there may be
data inconsistency due to a conflict because a change that wa he message queue
is contradictory to a subsequent change by the user.  In thi bitflag where all
conflicts are merged and it is up to app developer to fig o. */
public list get_friends(user1){
    list actual_friends = new list();
    list friends = get_friends();      foreach ds){          if(friend.status ==
friendstatus.confirmed){ //no conflict
        actual_friends.add(friend); status &= friendstatus.confirmed)
            and !(friend.status &= leted)){         // assume friend is
confirmed as long as it wasn't
        friend.status = friend ed;
        actual_friends.ad
        update_friend nd, status.confirmed);       }else{ //assume deleted
if there is a confli
        update_frie r1, friend, status.deleted)
    }     }//foreach actual_friends;
}
```

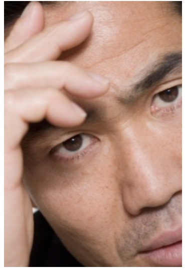
But inconsistenc

My rent check bounced?
That can't be right!

- Inconsistency causes bugs
  - Clients would never be able to trust servers… a free-for-all



Tommy Tenant
Anytown, USA
0000
Jason Fane Properties
$1150.00
INSUFFICIENT FUNDS
Sept 2009
Tommy Tenant
0000000 000 000 0 0000



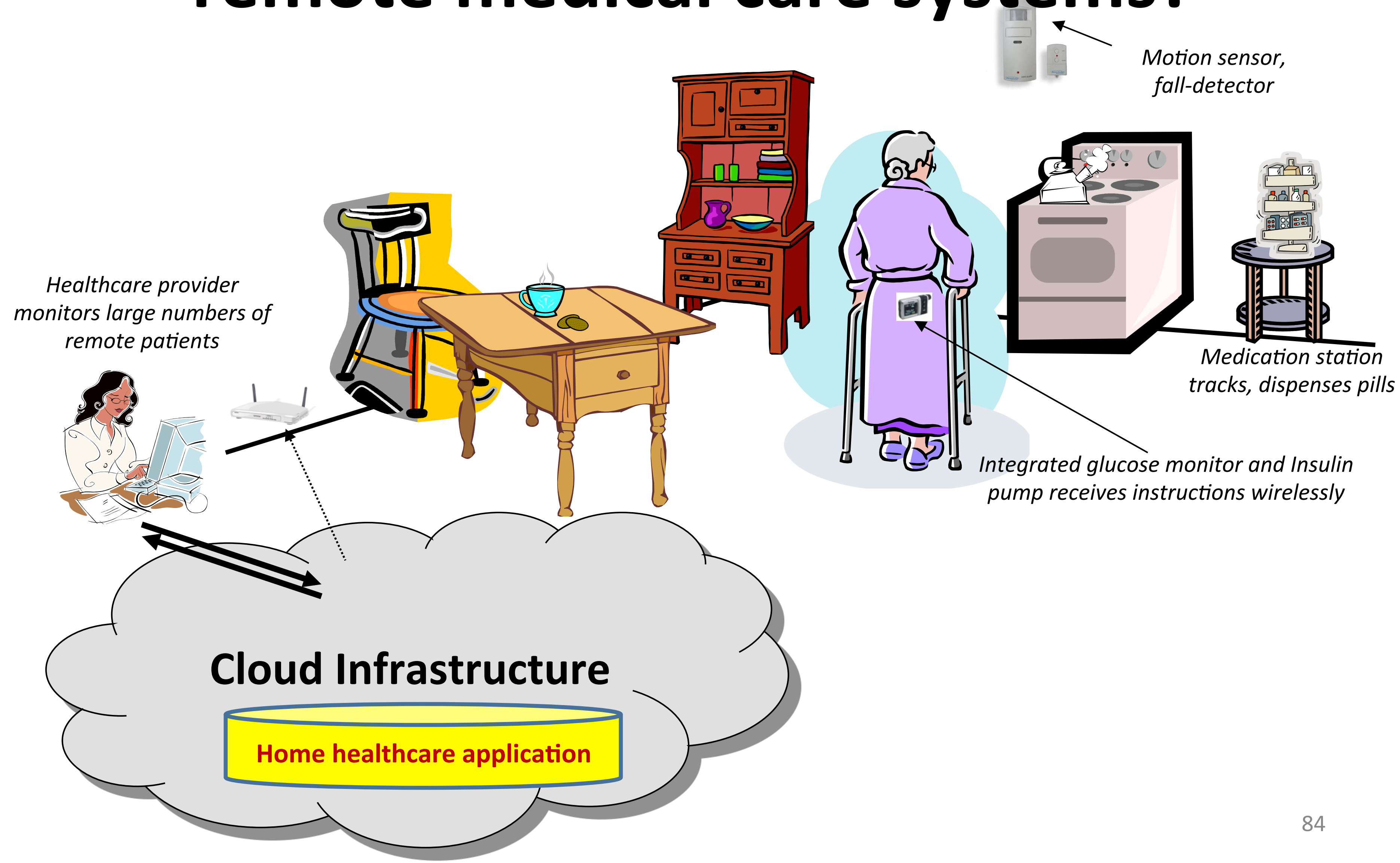
- Weak or "best effort" consistency?
  - Strong security guarantees demand consistency
  - Would you trust a a bank that offered "weak consistency" that may occasionally lose your money for better scalability?

What does “consistency” mean?

- As used in CAP, consistency is about two things
  - First, that updates to the same data item are applied in some agreed-upon order
  - Second, that once an update is acknowledged to an external user, it won’t be forgotten

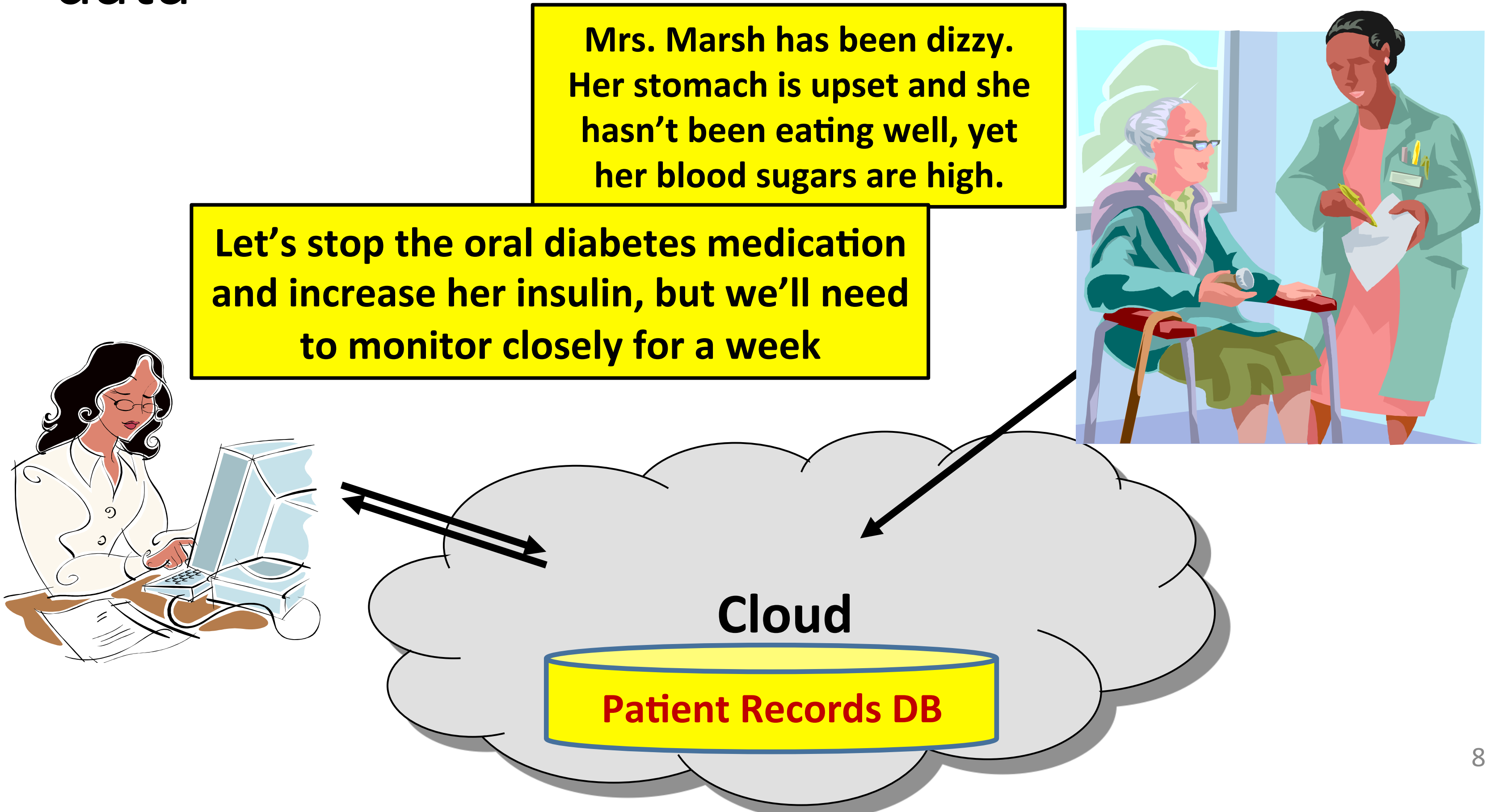
- Not all systems need both properties

What properties are needed in remote medical care systems?


Motion sensor,
fall-detector
Healthcare provider
monitors large numbers of
remote patients
Medication station
tracks, dispenses pills
Integrated glucose monitor and Insulin
pump receives instructions wirelessly
Cloud Infrastructure
Home healthcare application

Which matters more: fast response, or durability of the data being updated?

- Need: Strong consistency and durability for data

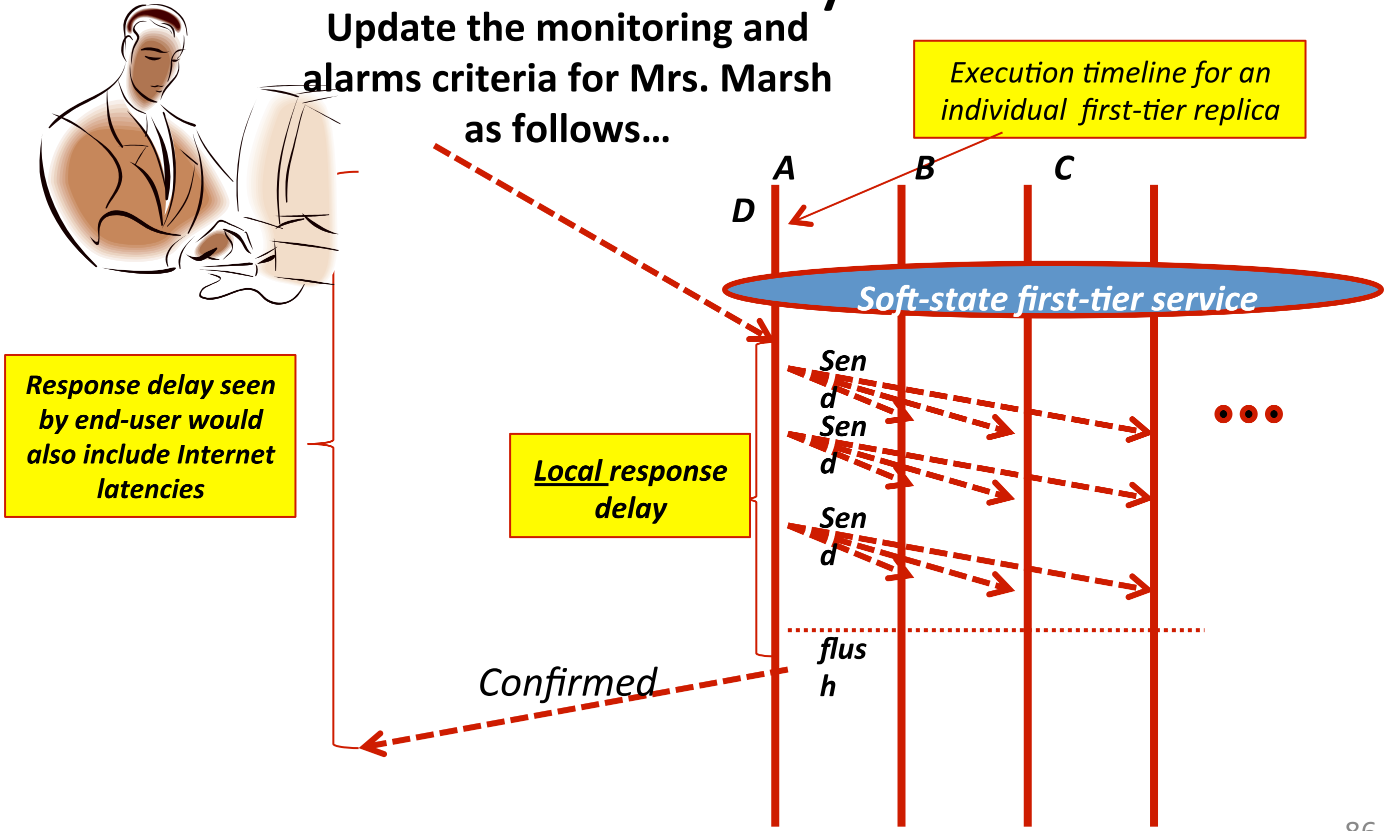


Mrs. Marsh has been dizzy. Her stomach is upset and she hasn't been eating well, yet her blood sugars are high.

Let's stop the oral diabetes medication and increase her insulin, but we'll need to monitor closely for a week

What if we were doing online monitoring?

- An online monitoring system might focus on real-time response and value consistency, yet be less concerned with durability



Update the monitoring and alarms criteria for Mrs. Marsh as follows…

Execution timeline for an individual first-tier replica

Response delay seen by end-user would also include Internet latencies

***Local** response delay*

Why does monitoring have weaker needs?

- When a monitoring system goes “offline” the device turns a red light or something on.
  - Later, on recovery, the monitoring policy may have changed and a node would need to reload it
  - Moreover, with in-memory replication we may have a strong enough guarantee for most purposes
- Thus if durability costs enough to slow us down, we might opt for a weaker form of durability in order to gain better scalability and faster responses!

This illustrates a challenge!

- Cloud systems just can't be approached in a one-size fits all manner

- For performance-intensive scalability scenarios we need to look closely at tradeoffs
  - Cost of stronger guarantee, versus
  - Cost of being faster but offering weaker guarantee

- If systems builders blindly opt for strong properties when not needed, we just incur other costs!
  - Amazon: Each 100ms delay reduces sales by 1%!

Key takeaway?

- Cloud apps often do not need full consistency.
  - Amazon and eBay do well with weak guarantees because many applications just didn't need strong guarantees to start with!
  - By embracing their weaker nature, we reduce synchronization and so get better response behavior
- Many NoSQL system target such apps – they provide high scalability, low latency, high availability, but compromise consistency.
- However, as cloud is more widely adopted, all the strong guarantees that drove decades of data management research and product development will be needed
  - E.g., safety critical applications
- Cloud will remain a platform for systems innovation & research for a foreseeable future.

Database Systems 101

Database Management Systems

- Systems for management of large volumes of data.
- Key features
  - Data model + High level query language (SQL)
  - Efficient storage and associative data access
  - Transactions -- ACID properties

Data Model

- Data model provides a level of abstraction between how data is physically stored versus how it is used by applications.
  - Hides complexity from application developers
- Relational model:
  - Data consists of tables, each consisting of a set of rows of the same type.
  - SQL language for accessing and manipulating data.
  - SQL compiled/interpreted into a query plan that corresponds to a tree of relational operators (select, project, join, group-by, aggregation, etc.).

Relational Database Systems: key technologies

- Record Manager: stores relations as a sequence of records
  - Variable length, repeated fields, small/large, schema updates, …
- Query Processor: implements basic relational operators (select, project, joins, aggregation, etc.) as well as ways to evaluate a query tree consisting of such operators.
- Query Optimizer: Takes an initial query tree and modifies it into an optimal execution plan often based on statistics maintained about relations
- Index Structures: data structures to support fast associative access to data (can be disk or memory resident)

Classification of Indexes

What do index entries point to?

- Dense: pointers to records
- Sparse: pointers to blocks

What is the order of *data records* being indexed?

- Unclustered: Data records are in no particular order
- Clustered: Order of data records corresponds to order of index keys

Classification of Indexes

	Clustered	Unclustered
Dense	Possible	Possible
Sparse	Possible	Does not make sense. Why?

Can only have one clustered index per relation. Why?

Example Indexes

Search Trees, e.g. B+Tree:

- Ordered by key
- Speed up =, <, >, <=, >=
- Clustered/unclustered and dense/sparse
- First few levels (e.g. 2) can usually fit in memory

Hash-Based, e.g. Linear Hashing:

- Unordered, pointers placed into buckets according to hash-function of the key
- Speed up =
- Does clustered/sparse make sense?

Transactions

- Transaction = consistent and failure resilient execution of database applications
  - E.g., withdraw $100 from a bank using ATM
- Transactions can be viewed as a computation that transforms the state of the database
  - State before Xaction: account balance $500, no money dispensed by ATM
  - State after Xaction: account balance $400, ATM dispenses $100
- Consistent transformation: ATM dispenses exactly the same money deducted from account
- Failure resilient: if money dispensed, it is debited from account – despite ALL failures.

ACID Properties of Xactions

- Atomicity: “all or nothing property”. No partial effects
  - Account debited if and only if money dispensed.
- Consistency: transformation caused by Xaction is “correct” /legal – satisfies database constraints.
  - Account after debit is still positive
- Isolation, aka serializability: partial effects of transactions hidden from other concurrent transactions.
- Durability: once transaction executes successfully (i.e., commits) its effects are permanent despite all failures

What Xactions provide to Users

- Simple failure semantics –
  - Either ATM machine dispenses the money or my account is not debited.
- Isolated view of the world –
  - No one else is simultaneously viewing my account.
- Provides an environment for developing complex concurrent (possibly distributed) applications wherein application writers do not worry about failures or concurrency – the system does so automatically!
- Possibly one of "the" most powerful ideas in distributed computing with widespread impact!
- Lots of extensions to basic transaction model – but that will take us too far away!
  - Xactions with savepoints, persistent savepoints, multilevel Xactions, nested Xactions, compensating Xactions, sagas, …

Application Writer's view of Xactions

```
BEGIN
     OK = make-airline-reservation(…)
     if (OK)
            OK = make-car-rental-reservation(…)
    else   ABORT
    if (OK)
                    OK = make-hotel-reservation(…)
   else ABORT
    if (OK)
           COMMIT
  else ABORT
END
```

- System ensures effects of committed transactions persistent, and aborted transactions are undone automatically.

Transactions & Cloud Data Management

- Transaction technology has long been used in enterprise world to implement consistent, durable, and atomic access to data.
  - In centralized, distributed, replicated databases.
- Can we not use transaction technology to implement consistent replicated data stores in the cloud?
- Short answer: perhaps, we can – specially when strong consistency guarantees are needed. But transactions incur huge costs and such costs not justified for many applications for which weak consistency guarantees suffice.

How are Xactions implemented?

- Concurrency Control Protocols to ensure serializability
  - Locking protocols
  - Timestamp based
  - Validation mechanisms
- Recovery Algorithms (abort, system restart, media restart, disaster recovery)
  - Shadow paging
  - Log-based recovery
    - Write ahead logs, redo/undo logging
- Atomic Commit Protocols in case transactions access multiple resource managers (e.g., as in distributed systems)
  - Two-phase commit, three phase commit protocols

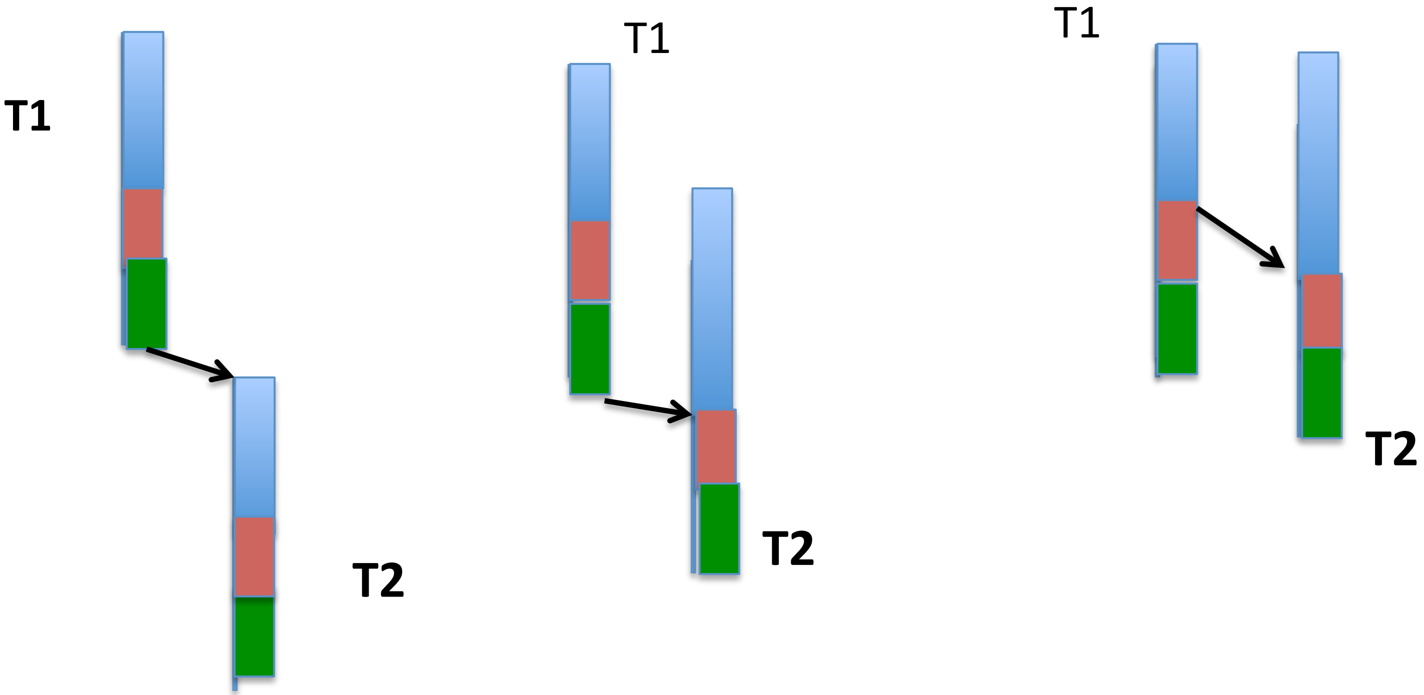
Locking

- Xaction acquires a lock on data item X before accessing X.
  - S lock for reads and X lock for updates
- Lock is intuitively similar to semaphores – no two transaction can hold (conflicting) locks on the same data at the same time.
- After lock is acquired, Xaction can read/write the data.
- Two Phase Locking: Xaction MUST acquire all locks before it releases any lock
- Two Phase locking ensures serializability – equivalence to some sequential execution of applications running as transactions.
- Lots of challenges
  - granularity of locks (field, tuple, subset of tuples possibly defined using a predicate, file, relation, ..)
  - How are locks implemented? In a centralized system? In a distributed system?
  - How do lock based protocols ensure serializability? What exactly is serializability?

Optimistic Concurrency Control

- Each transaction consists of three phases
  - Read phase: transactions read data from database. Updates local copy.
  - Validation phase: check if transaction can apply its updates to database without causing inconsistency
  - Write phase: copy its changes to database.
- Validation guarantes that if Ti < Tj, then one of these conditions hold
  - Ti completes write phase before Tj starts read phase
  - Write-set(Ti) intersect read-set(Tj) = 0 and write-phase of Ti finishes before Tj starts its write phase.
  - Both write-set(Ti) intersect read-set(Tj) = 0 and write-set(Ti) intersect read-set(Tj) = 0 and Ti finishes Tj finishes its read phase before Tj finishes its read phase

Optimistic Concurrency Control



T2 can validate itself if

- T1 fully executed before T2 started, or
- T1 has finished its write phase, and T2 did not read any data written by T1, or
- T1 has finished its read phase, and T2 did not read anything written by T2 or overwritten anything written by T2

So what do the Concurrency Control Mechanisms Guarantee?

- **Serializability** (aka isolation) – equivalence to sequential execution of the transactions.
  - No transactions sees partial effects of any other concurrently executing transaction.
- Imagine two transactions:
  - T1: transfer $100 from account A to B (initial values 1000 each)
  - T2: Read A and B to determine total.
- Non-Serializable execution: T2 reads state of A after debit by T1, but state of B before credit by T1 (will see $100 dissapear!)
  - Locking prevents it – T2 will not get a lock on A, and hence will have to wait until T1 releases lock (which it will after credit).
  - Optimistic protocol prevents it -- if read phase of T1 and T2 overlapped, T2's validation will fail.

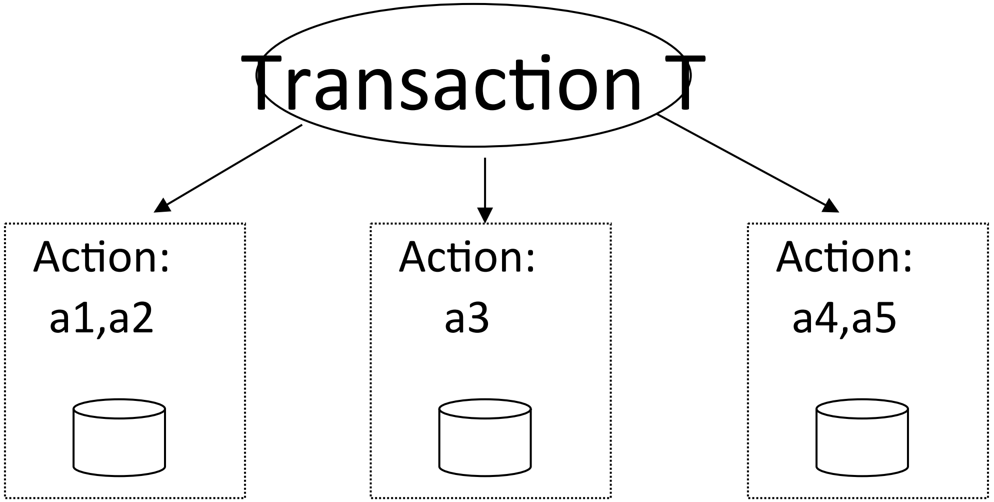
Logging

- Consider an application transferring $100 from Account X to Y.
- Let initial values be X = 1000, Y = 2000
- After execution of application X = 900, Y = 2100
- Since transaction running the application may abort, the system must save enough state to reconstruct old values (i.e., 1000, 2000)
  - One approach: prevent news values to reach the disk. In case of abort, invalidate memory buffers, old values still on disk!
  - Alternate approach: log before values X = 1000, Y = 2000. These logs called UNDO logs.
  - UNDO logs satisfy the requirement
    - UNDO log of x (y) + new value of x (y) = old value of x (y)
- If a Xaction commits, we MUST be able to ensure durability despite failures.
  - One approach: push changes to stable storage prior to COMMIT (force policy)
  - Alternate approach: log after values X = 900, Y = 2100. Make REDO logs durable prior to commit. Use logs to recover state in case of failures
- Why is logging potentially a better strategy?
- But lots of complications – how are logs implemented, what do we log (e.g., physical before/after values, operations?), are there interplay between how we log and what we lock? … all complexities are hidden under the rug for now.

Write Ahead Logging

- Redo logs made stable prior to commit. What about Undo logs?
- Undo logs used to remove effects of partially executed transactions.
- If machine fails, stable copy of database can have effects of partially executed transactions.
- Undo logs must to made stable prior to modified objects being written to stable database (Write ahead logging!)

Atomic Commit Protocol



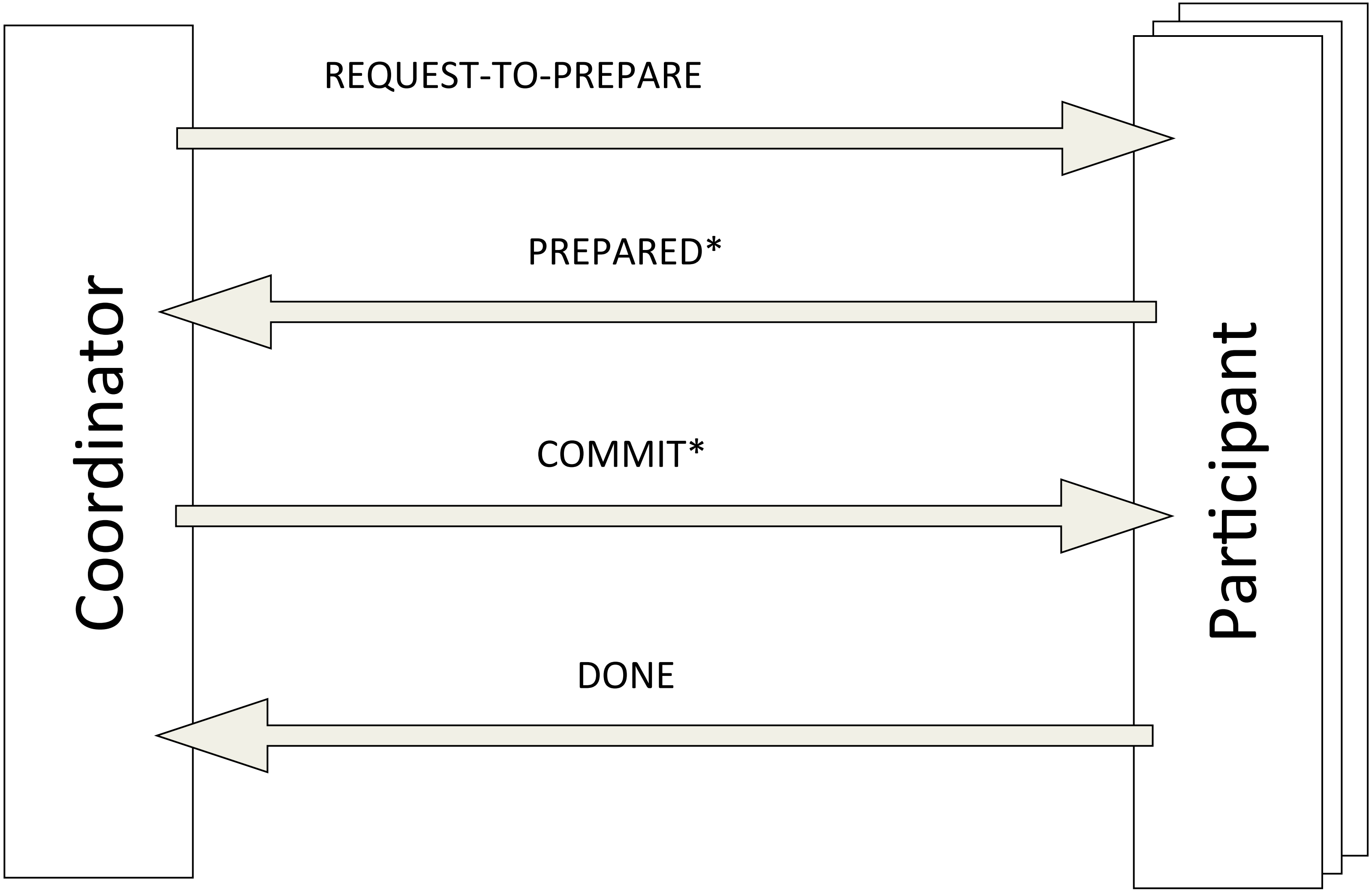
- Atomic commit protocol must ensure that despite failures, when failures are repaired, Xaction commits or aborts at all sites
- Notice slight difference in goals compared to consensus problem which required all non-failed site to reach an agreement.
- In consensus, focus was on progress of “live” processes. In ACP, focus is on consistent decisions. Non-failed processes (as we will see) may need to wait (indefinitely)

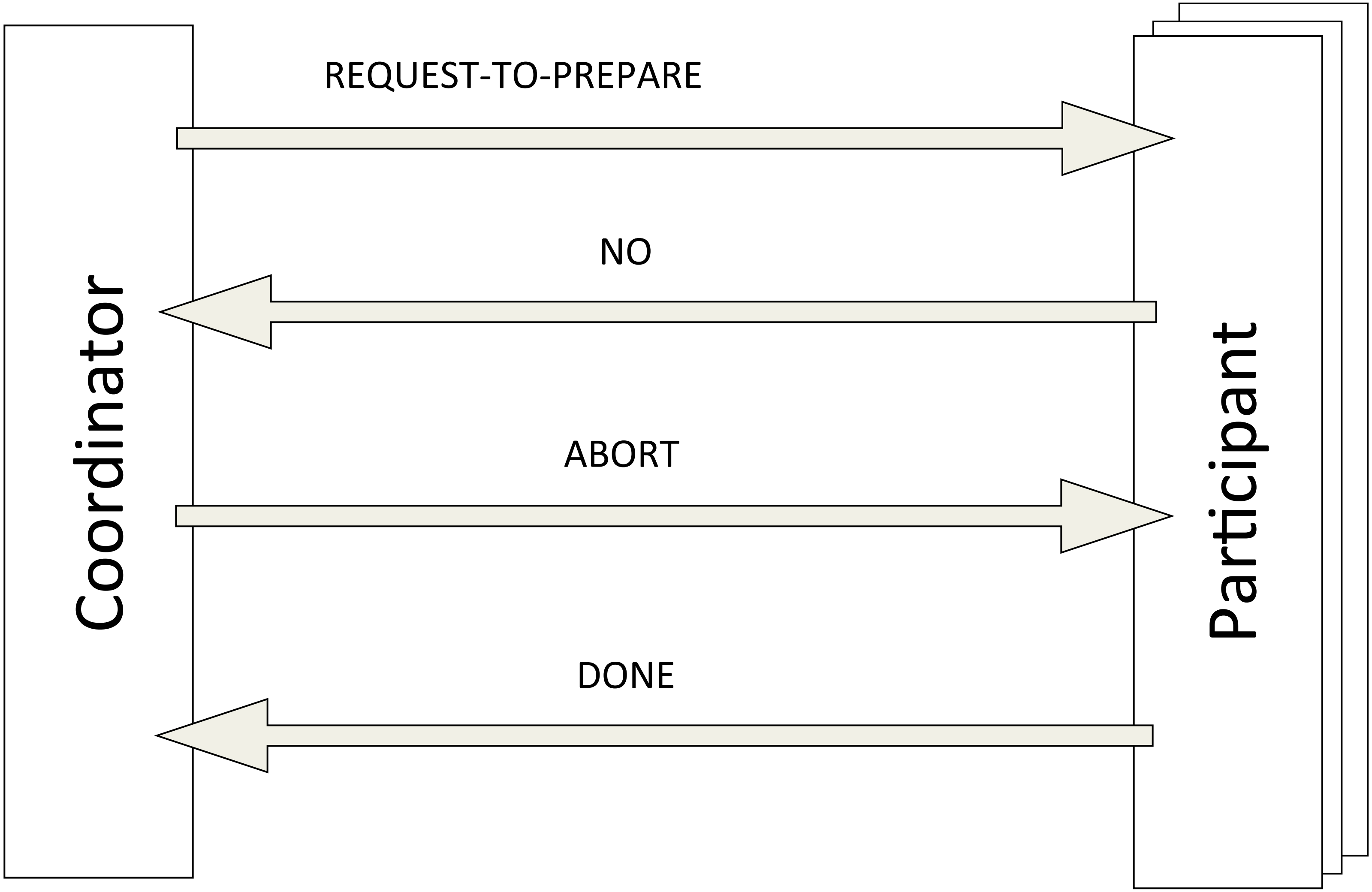
Common ACP Protocols

- Two phase commit (lots of variants)
  - Centralized
  - Distributed
  - Linear
  - Hierarchical
  - Transfer of commit
  - Group commit
  - With cooperative termination
  - …
- Three phase commit
  - Reduces blocking under certain situations but at additional overheads

Terminology

- Resource Managers (RMs)
  - Usually databases
- Participants
  - RMs that did work on behalf of transaction
- Coordinator
  - Component that runs two-phase commit on behalf of transaction


Coordinator
REQUEST-TO-PREPARE
PREPARED*
COMMIT*
DONE
Participant



Coordinator

REQUEST-TO-PREPARE

NO

ABORT

DONE

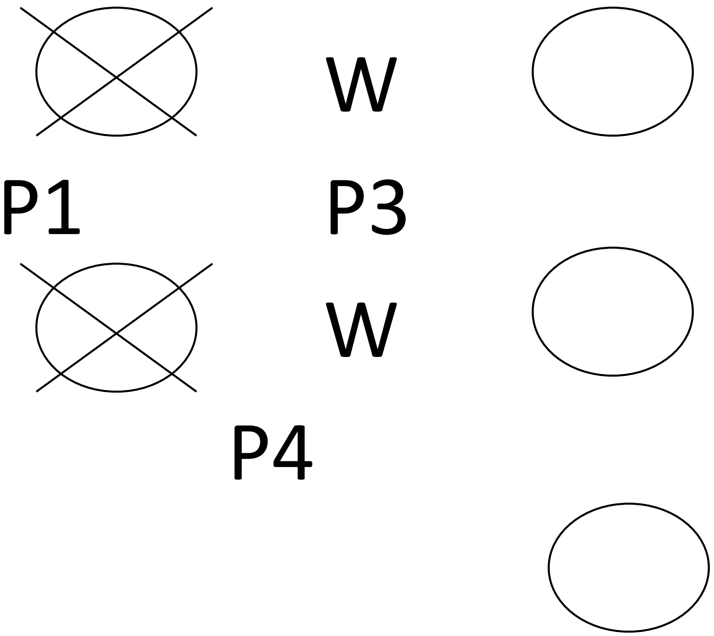
Participant

2PC (cont.)

- At various stages, either the coordinator or the participant wait for messages (to make progress).
- What if the message do not arrive?
  - Slow network, lost messages, failures.
  - Potential failures detected using timeout.
  - Time out actions to help make progress
- What if a process fails (coordinator or participant)
  - Recovery actions used to ensure atomic commit.

2PC is blocking
(blocking → a non-failed process may have to wait for an unbounded time waiting for another process to recover)

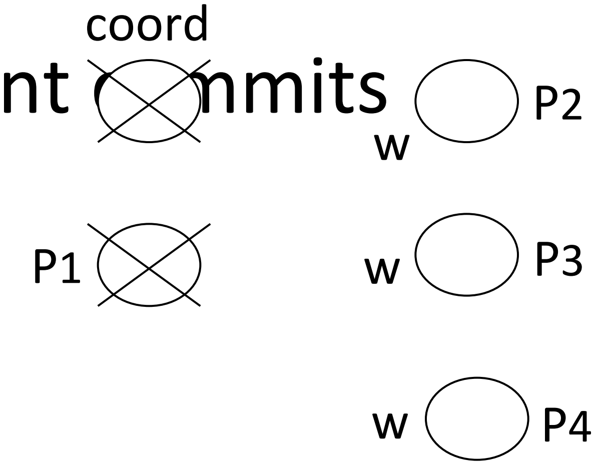
- Sample scenario:



- Coord P2
- W
- P1 P3
- W
- P4 W

- Case I:
- P1 → “W”; coordinator sent commits
- P1 → “C”
- Case II:
- P1 → NO; P1 → A

- ⇒ P2, P3, P4 (surviving participants)
- cannot safely abort or commit transaction

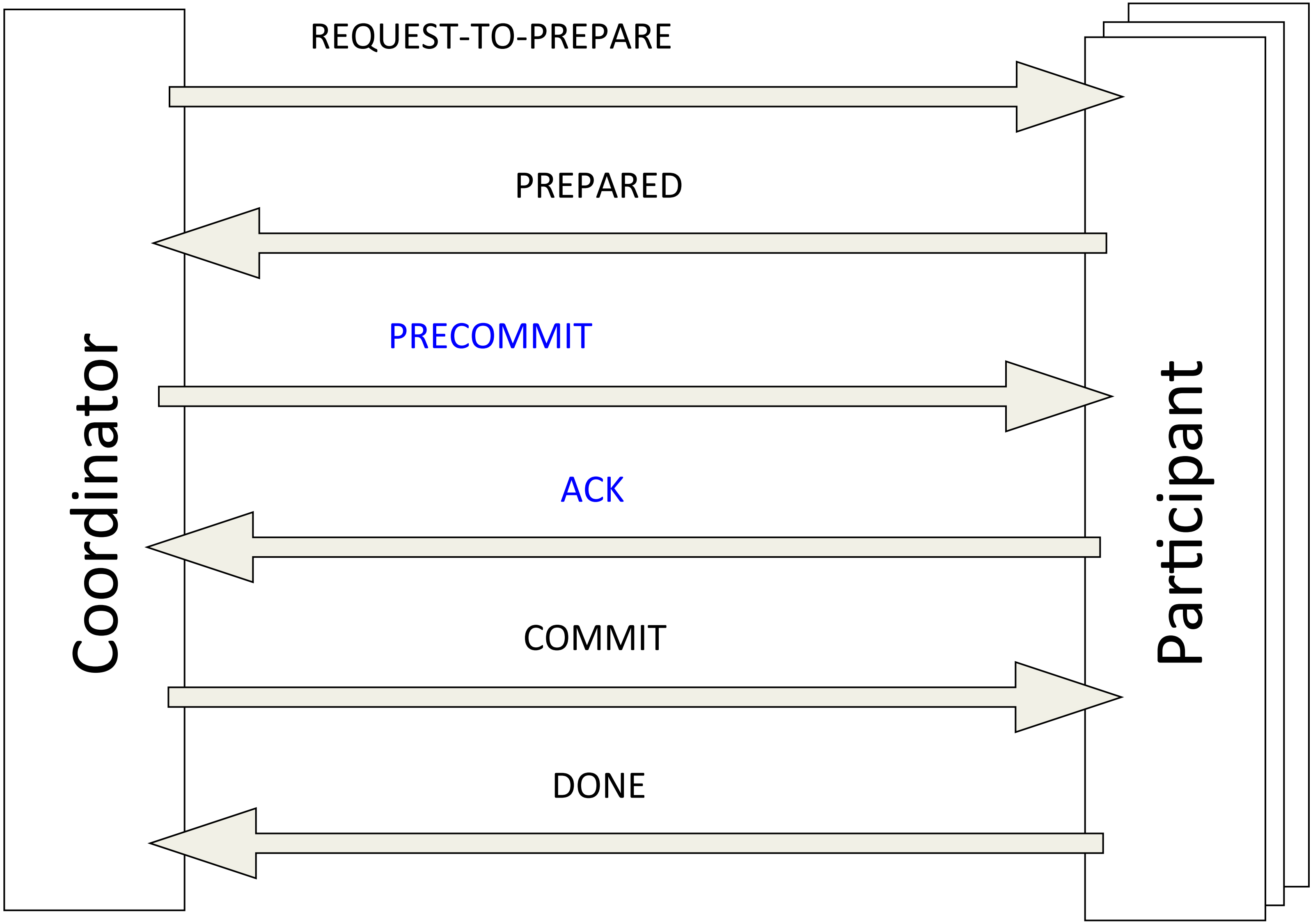


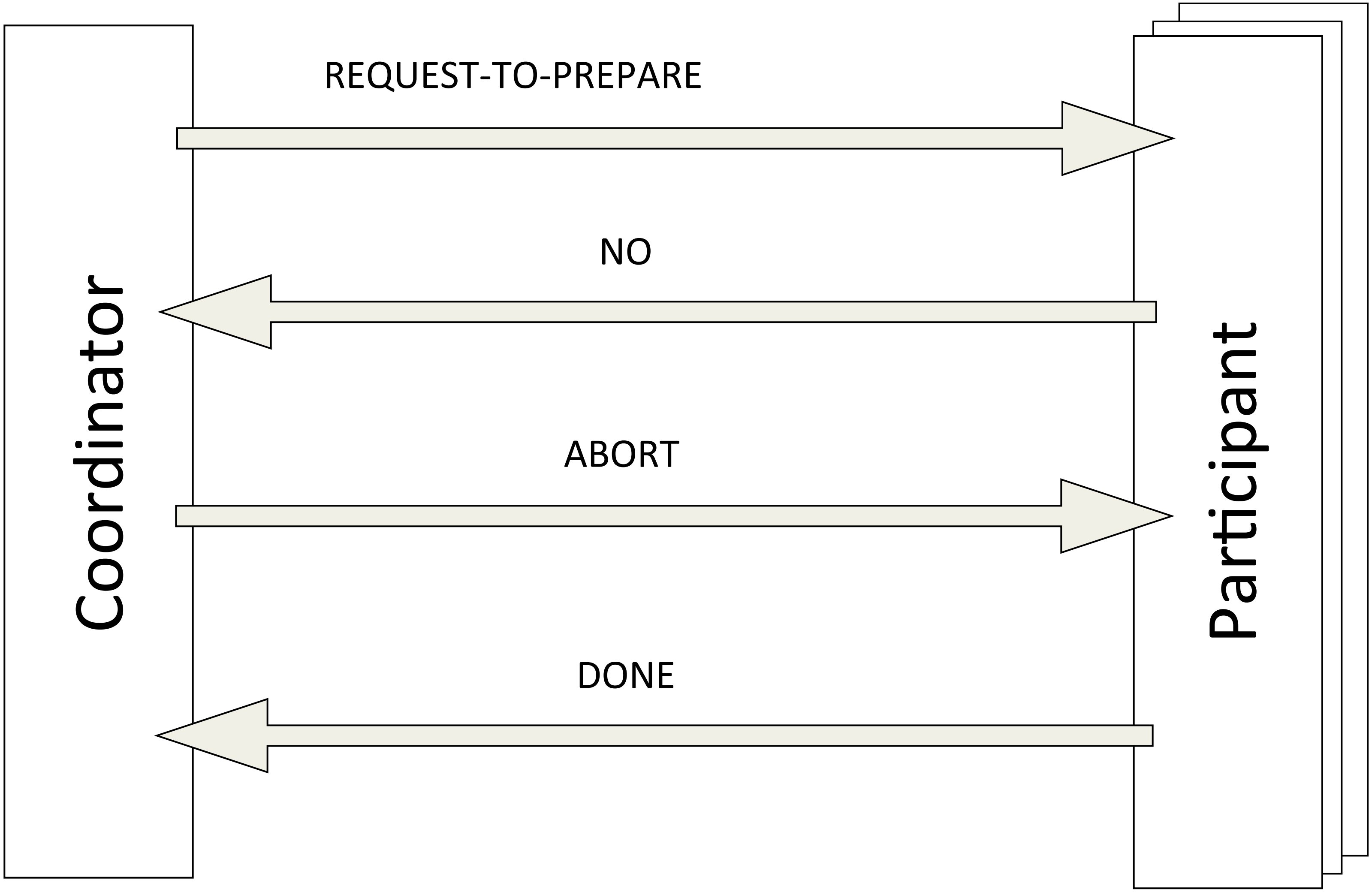
Is there a non-blocking protocol?

- Theorem:
  - If communications failure or total site failures (i.e., all sites are down simultaneously) are possible, then every atomic protocol may cause processes to become blocked.
- An exception:
  - if communication failures cannot occur, it is possible to design such a protocol (Skeen et. al. 83)

Approach to Making ACP Non-blocking

- For a given state S of a transaction T in the ACP, let the concurrency set of S be the set of states that other sites could be in.
- For example, in 2PC, the concurrency set of PREPARE state is {PREPARE, ABORT, COMMIT}
- We develop non-blocking protocol, we will
  - ensures that concurrency set of a transaction does not contain both a commit and an abort
  - There exists no non-committable state whose concurrency set contains a commit. A state is committable if occupancy of the state by any site implies everyone has voted to commit the transaction.
- Necessity of these conditions illustrated by considering a situation with only 1 site operational. If either of the above violated, there will be blocking.
- Sufficiency illustrated by designing a termination protocol that will terminate the protocol correctly if the above assumptions hold.


Coordinator
REQUEST-TO-PREPARE
PREPARED
PRECOMMIT
ACK
COMMIT
DONE
Participant



Coordinator

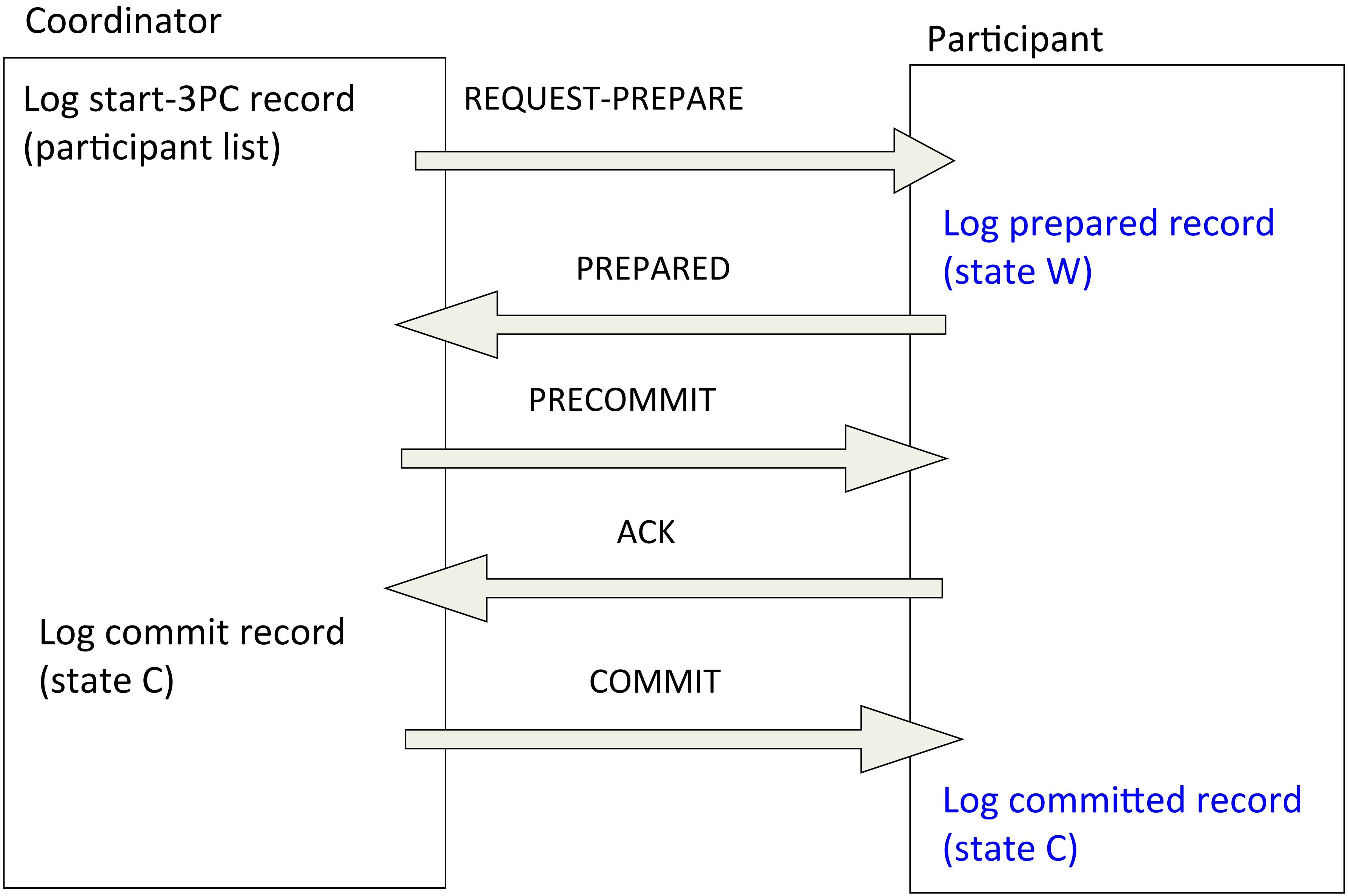
REQUEST-TO-PREPARE

NO

ABORT

DONE

Participant



Coordinator

Participant

Log start-3PC record
(participant list)

REQUEST-PREPARE

Log prepared record
(state W)

PREPARED

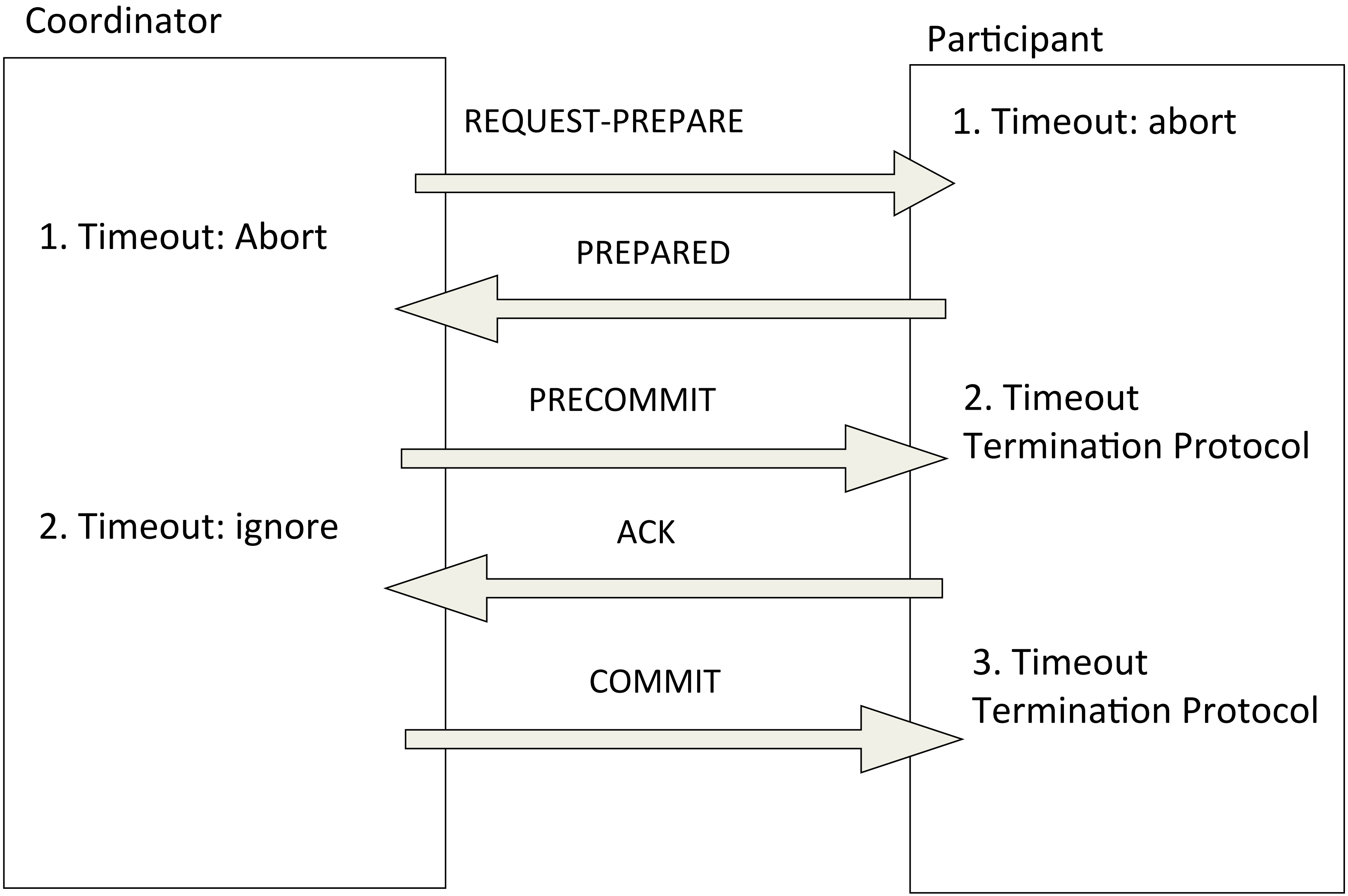
PRECOMMIT

ACK

Log commit record
(state C)

COMMIT

Log committed record
(state C)



Coordinator

Participant

REQUEST-PREPARE

1. Timeout: abort

1. Timeout: Abort

PREPARED

PRECOMMIT

2. Timeout
Termination Protocol

2. Timeout: ignore

ACK

COMMIT

3. Timeout
Termination Protocol

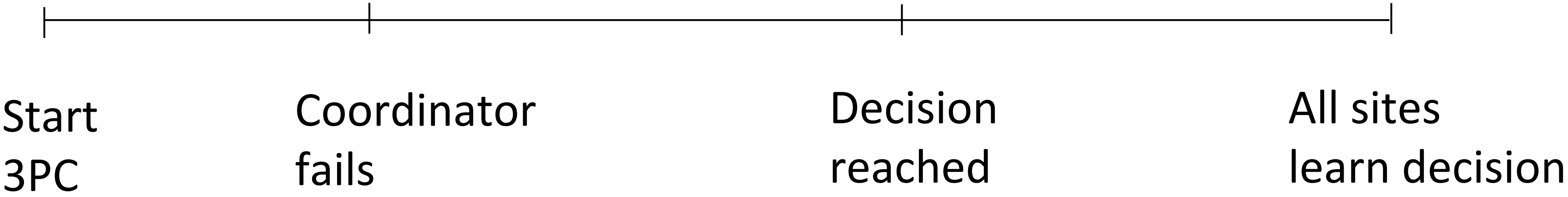
Process categories

- Three categories
  - Operational
    - Process has been up since start of 3PC
  - Failed
    - Process has halted since start of 3PC, or is recovering
  - Recovered
    - Process that failed and has completed recovery

Three Phase Commit - Termination Protocol

- Choose a backup coordinator from the remaining operational sites.
- Backup coordinator sends messages to other operational sites to make transition to its local state (or to find out that such a transition is not feasible) and waits for response.
- Based on response as well as its local state, it continues to commit or abort the transaction.
- It commits, if its concurrency set includes a commit state. Else, it aborts.

Termination Protocol



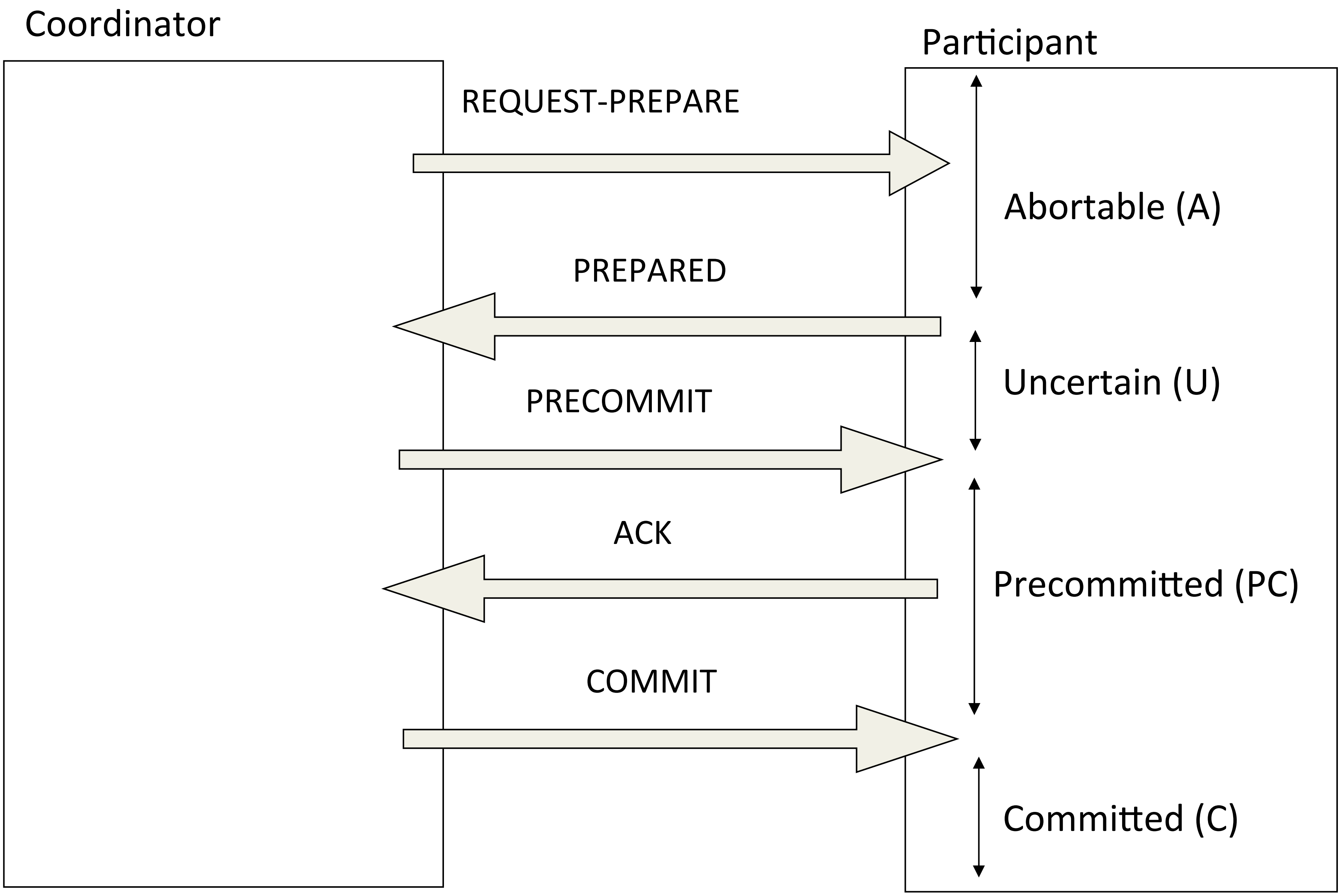
Start
3PC

Coordinator
fails

Decision
reached

All sites
learn decision

- Only operational processes participate in termination protocol.
- Recovered processes wait until decision is reached and then learn decision



Coordinator

Participant

REQUEST-PREPARE

Abortable (A)

PREPARED

Uncertain (U)

PRECOMMIT

ACK

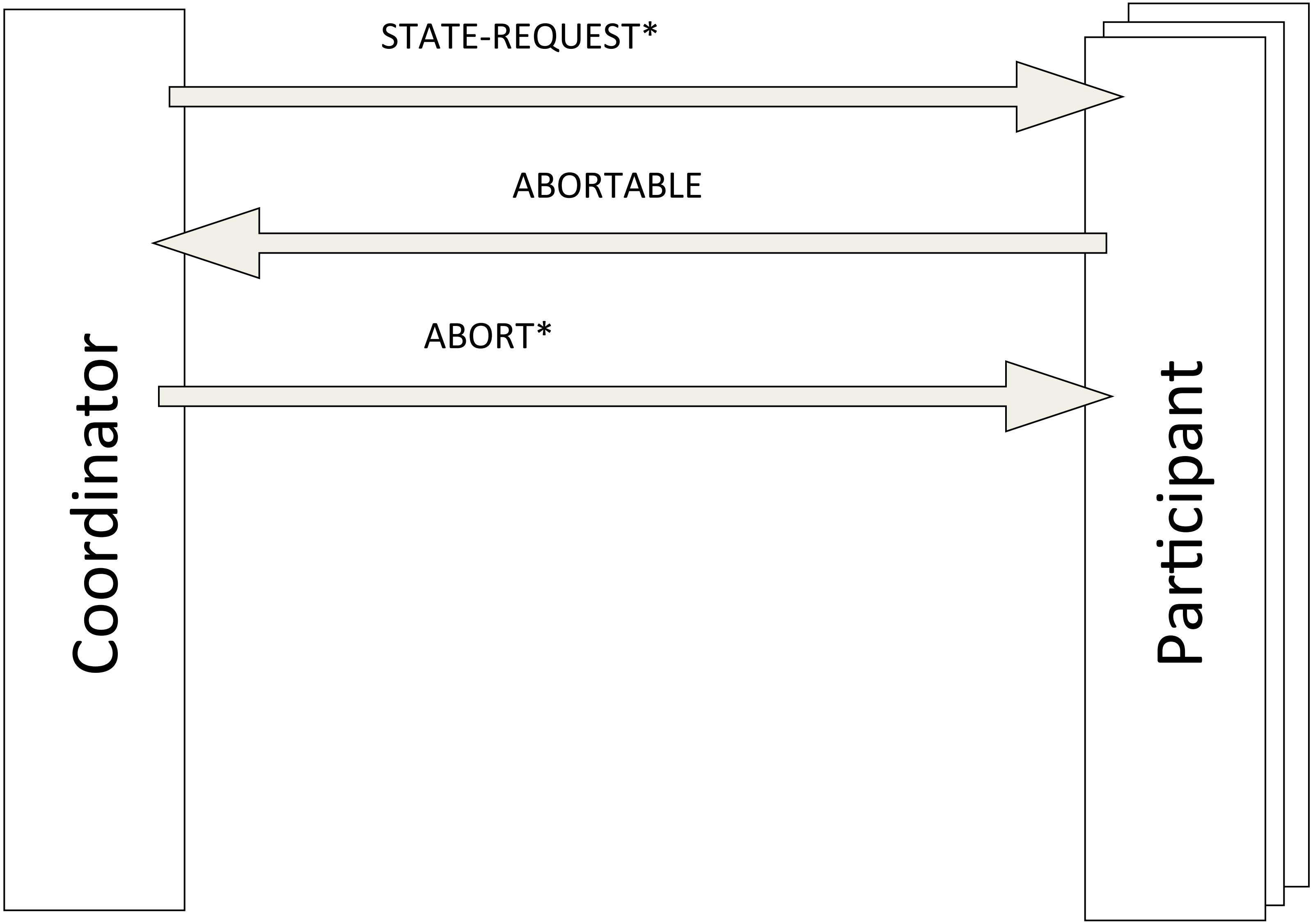
Precommitted (PC)

COMMIT

Committed (C)

Termination Protocol

- Elect new coordinator
  - Use Election Protocol (coming soon…)
- New coordinator sends STATE-REQUEST to participants
- Makes decision using termination rules
- Communicates to participants



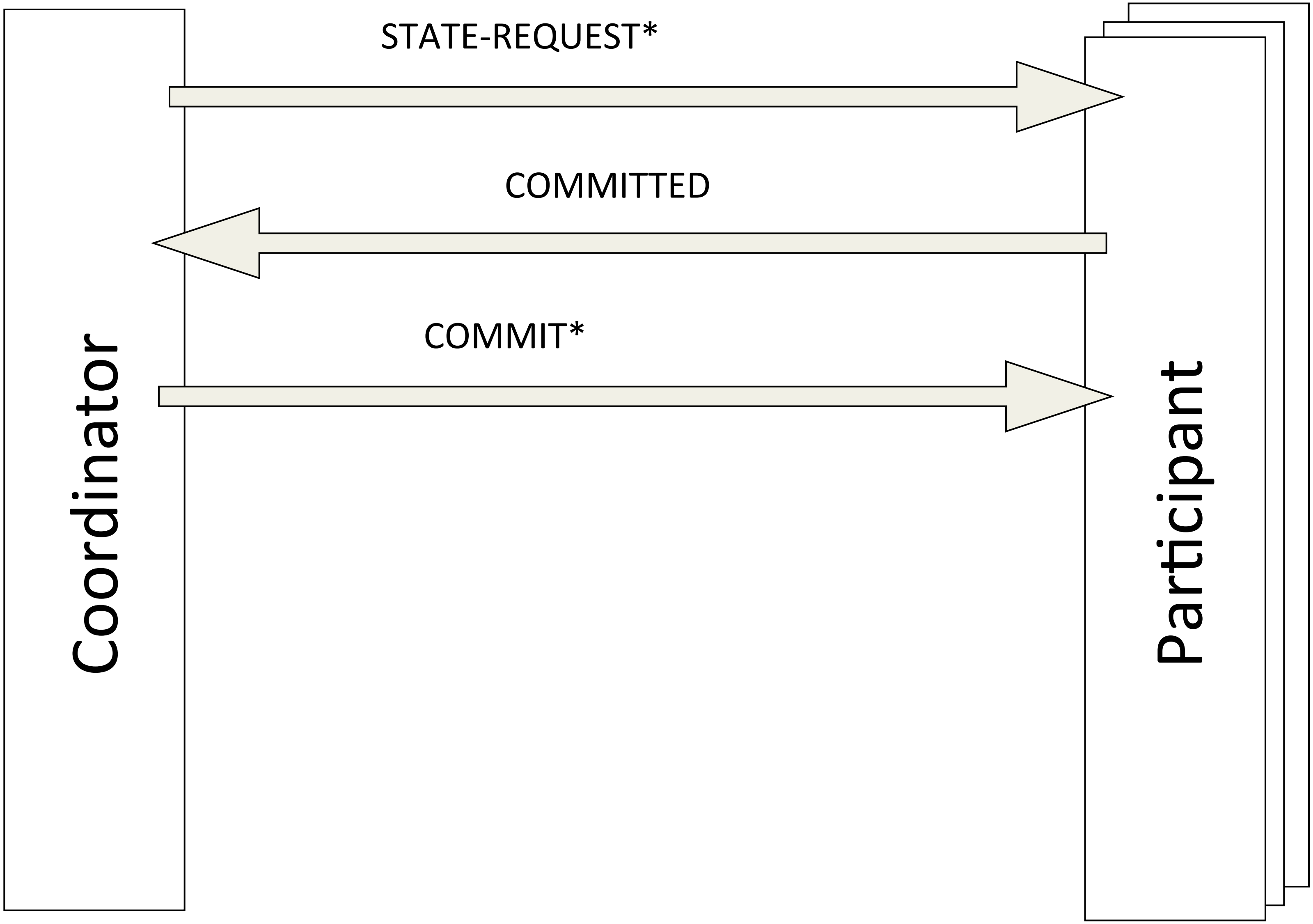
STATE-REQUEST*

ABORTABLE

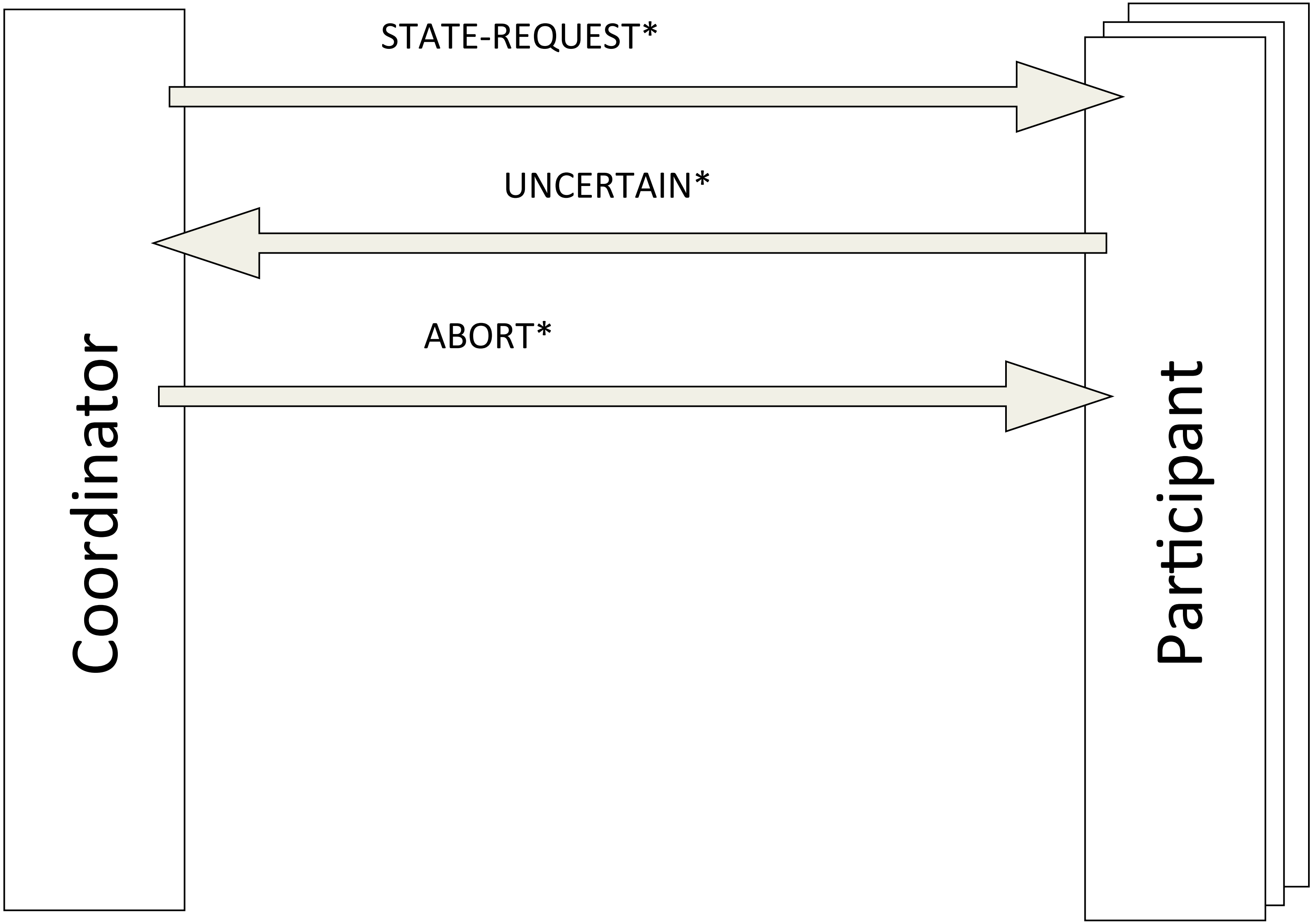
ABORT*

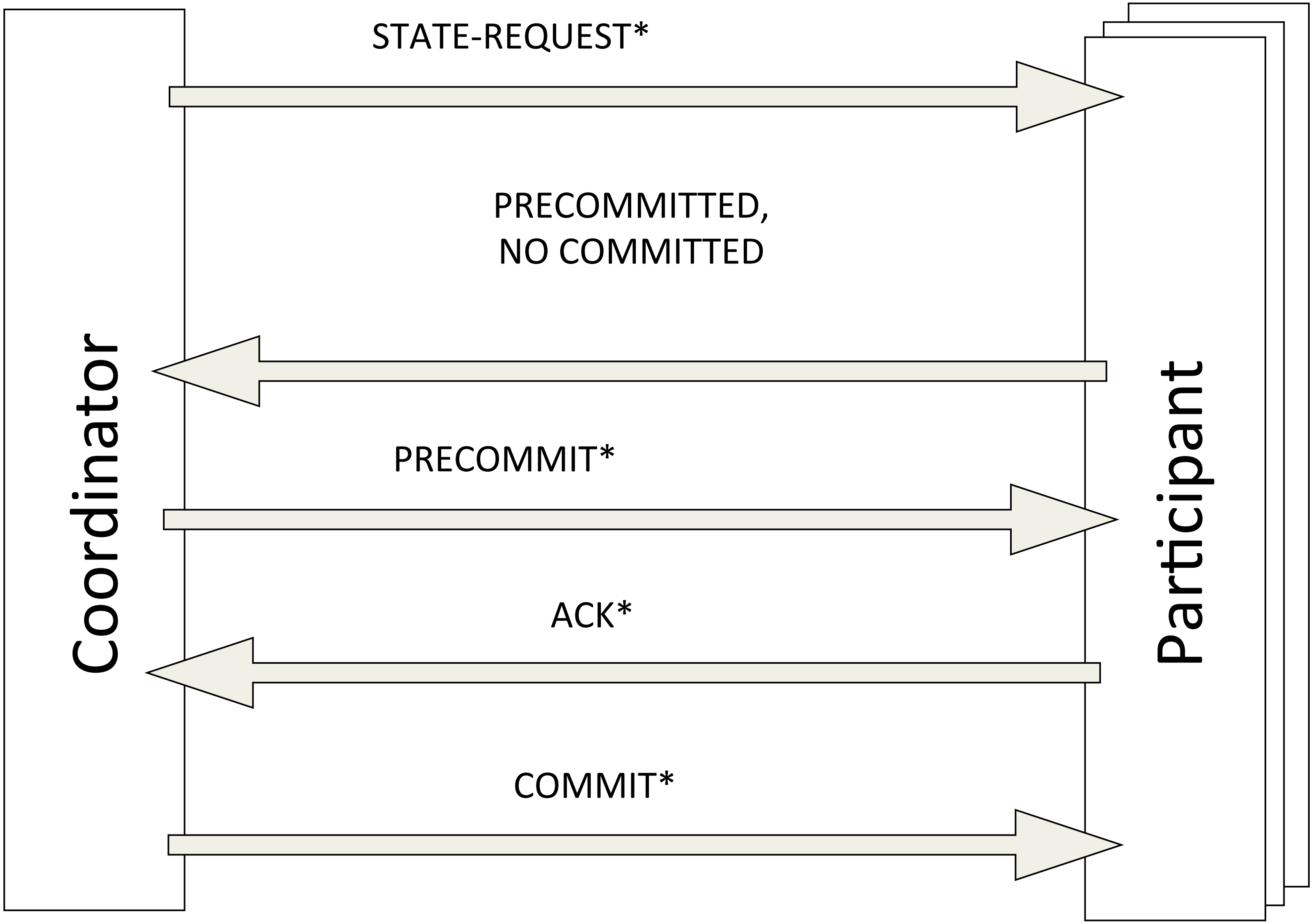
Coordinator

Participant



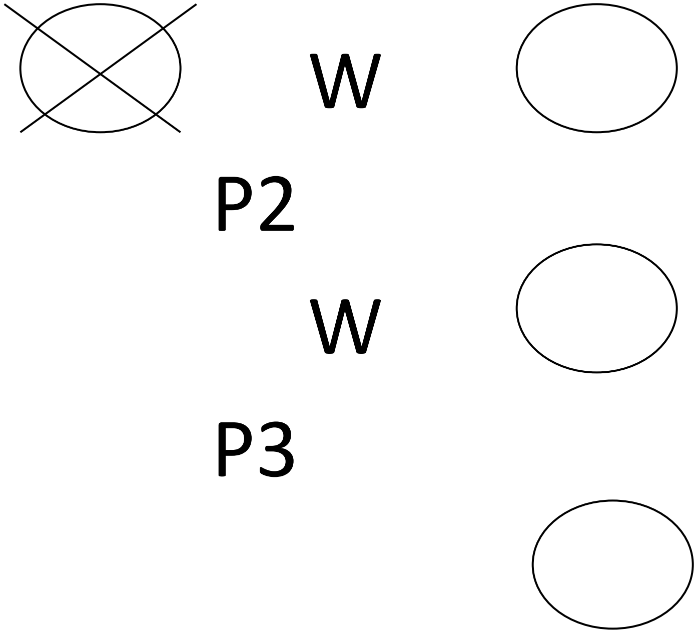
STATE-REQUEST*
COMMITTED
COMMIT*
Coordinator
Participant


STATE-REQUEST*
UNCERTAIN*
ABORT*
Coordinator
Participant


STATE-REQUEST*
PRECOMMITTED,
NO COMMITTED
PRECOMMIT*
ACK*
COMMIT*
Coordinator
Participant

Termination Protocol

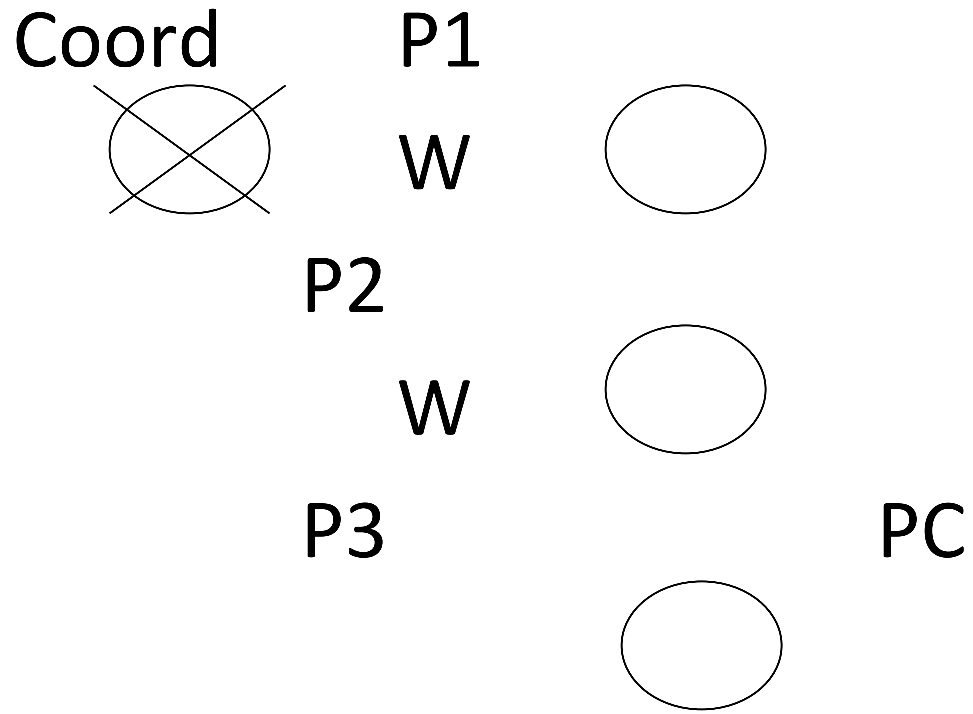
- Sample scenario:



- Coord P1
- W
- P2
- W
- P3 W

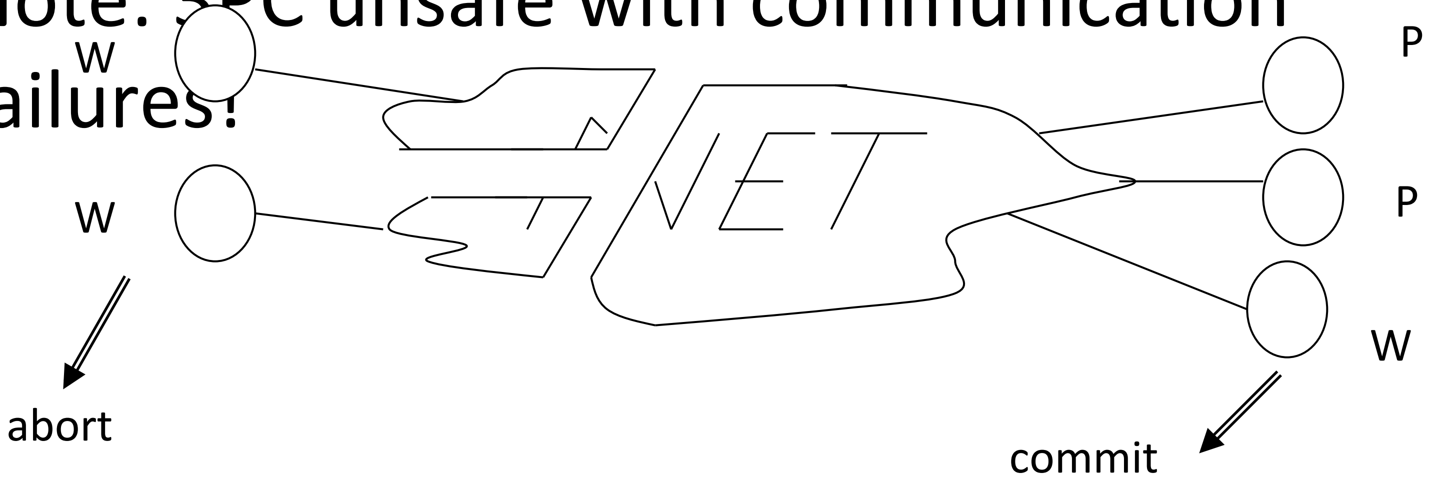
Termination Protocol

- Sample scenario:



- Coord P1
- W
- P2
- W
- P3 PC

- Note: 3PC unsafe with communication failures!



P2P Systems 101

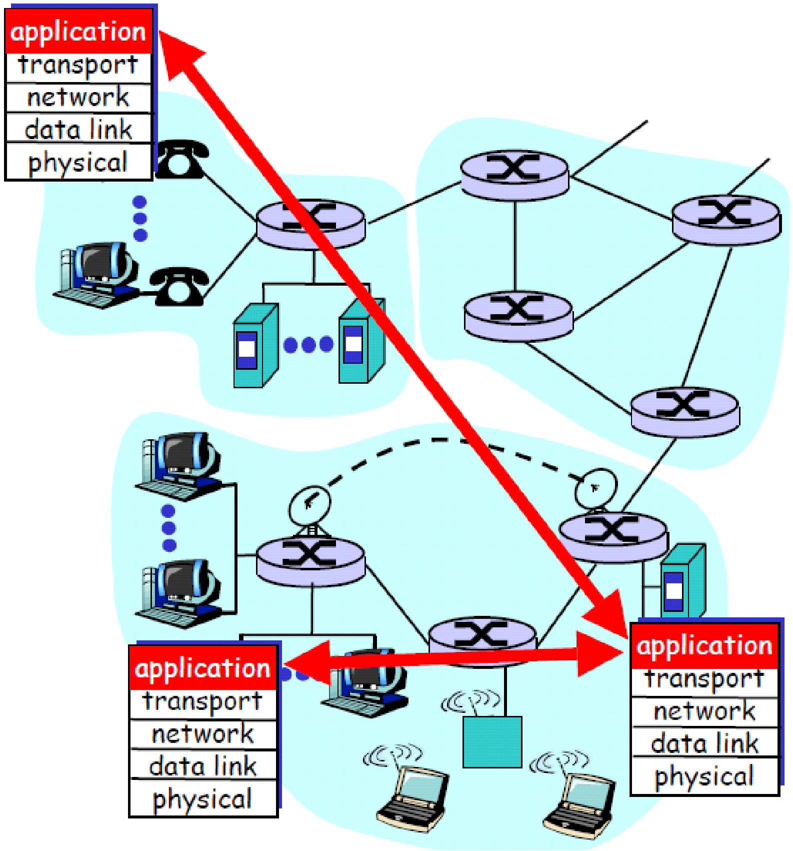
Use the vast resources of machines at the edge of the Internet to build a network that allows resource sharing without any central authority.

Characteristics of P2P Systems

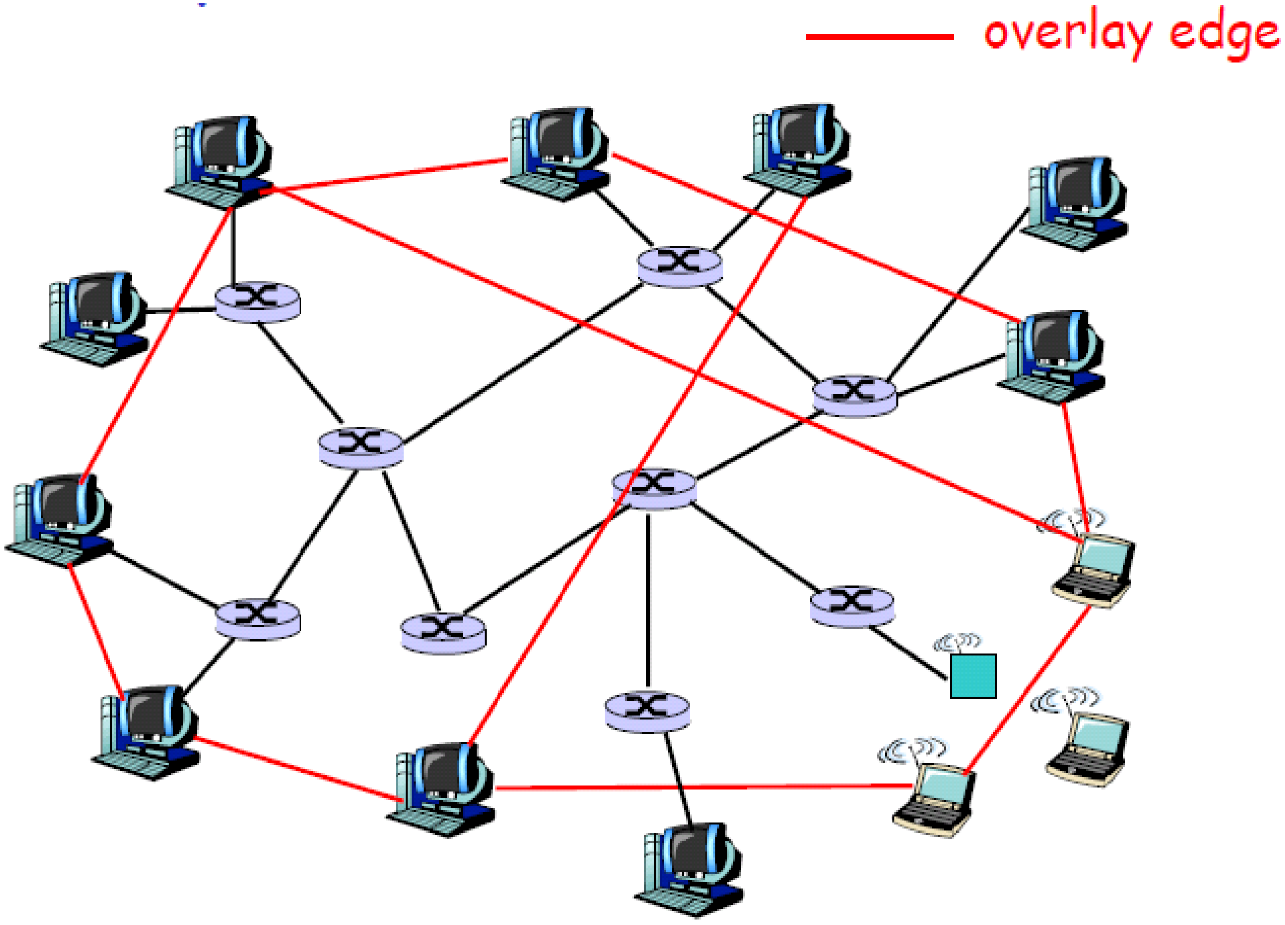
- **Exploit edge resources**.
  - Storage, content, CPU, Human presence.
- **Significant autonomy from any centralized authority.**
  - Each node can act as a Client as well as a Server.
- **Resources at edge have intermittent connectivity, constantly being added & removed.**
  - Infrastructure is untrusted and the components are unreliable.

Overlays : All in the application layer

- **Tremendous design flexibility**
  - Topology, maintenance
  - Message types
  - Protocol
  - Messaging over TCP or UDP
- **Underlying physical network is transparent to developer**
  - But some overlays exploit proximity



Overlay Network



A P2P network is an **overlay network**. Each link between peers consists of one or more IP links.

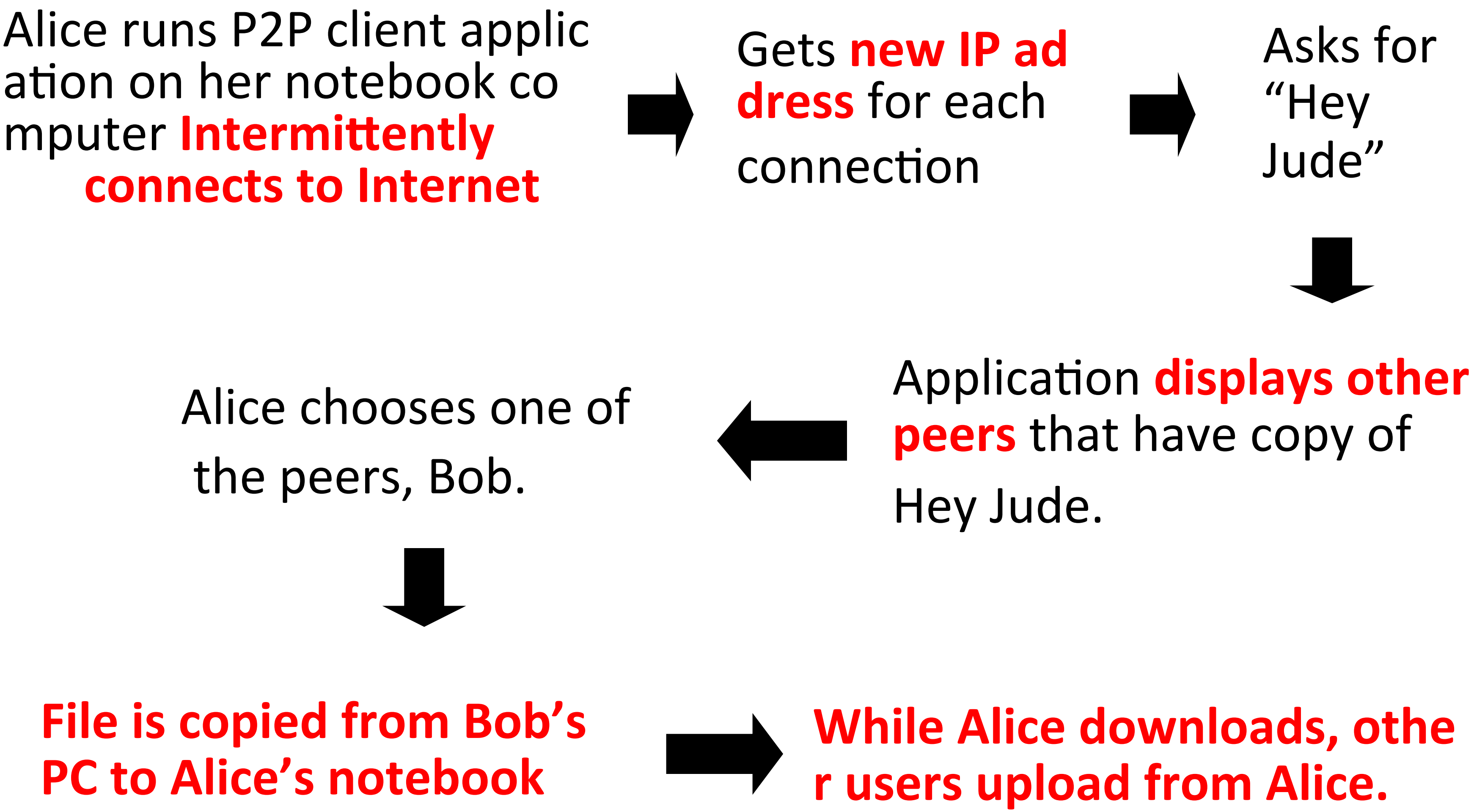
Overlay Graph

- **Virtual edge**
  - TCP connection
  - or simply a pointer to an IP address
- **Overlay maintenance**
  - Periodically ping to make sure neighbor is still alive
  - Or verify aliveness while messaging
  - If neighbor goes down, may want to establish new edge
  - New incoming node needs to bootstrap
  - Could be a challenge under high rate of churn
    - Churn : dynamic topology and intermittent access due to node arrival and failure

P2P Applications

- **P2P File Sharing**
  - Napster, Gnutella, Kazaa, eDonkey, BitTorrent
  - Chord, CAN, Pastry/Tapestry, Kademlia
- **P2P Communications**
  - MSN, Skype, Social Networking Apps
- **P2P Distributed Computing**
  - Seti@home

P2P File Sharing



Alice runs P2P client applic
ation on her notebook co
mputer **Intermittently**
connects to Internet

Gets **new IP ad**
dress for each
connection

Asks for
“Hey
Jude”

Application **displays other**
peers that have copy of
Hey Jude.

Alice chooses one of
the peers, Bob.

File is copied from Bob’s
PC to Alice’s notebook

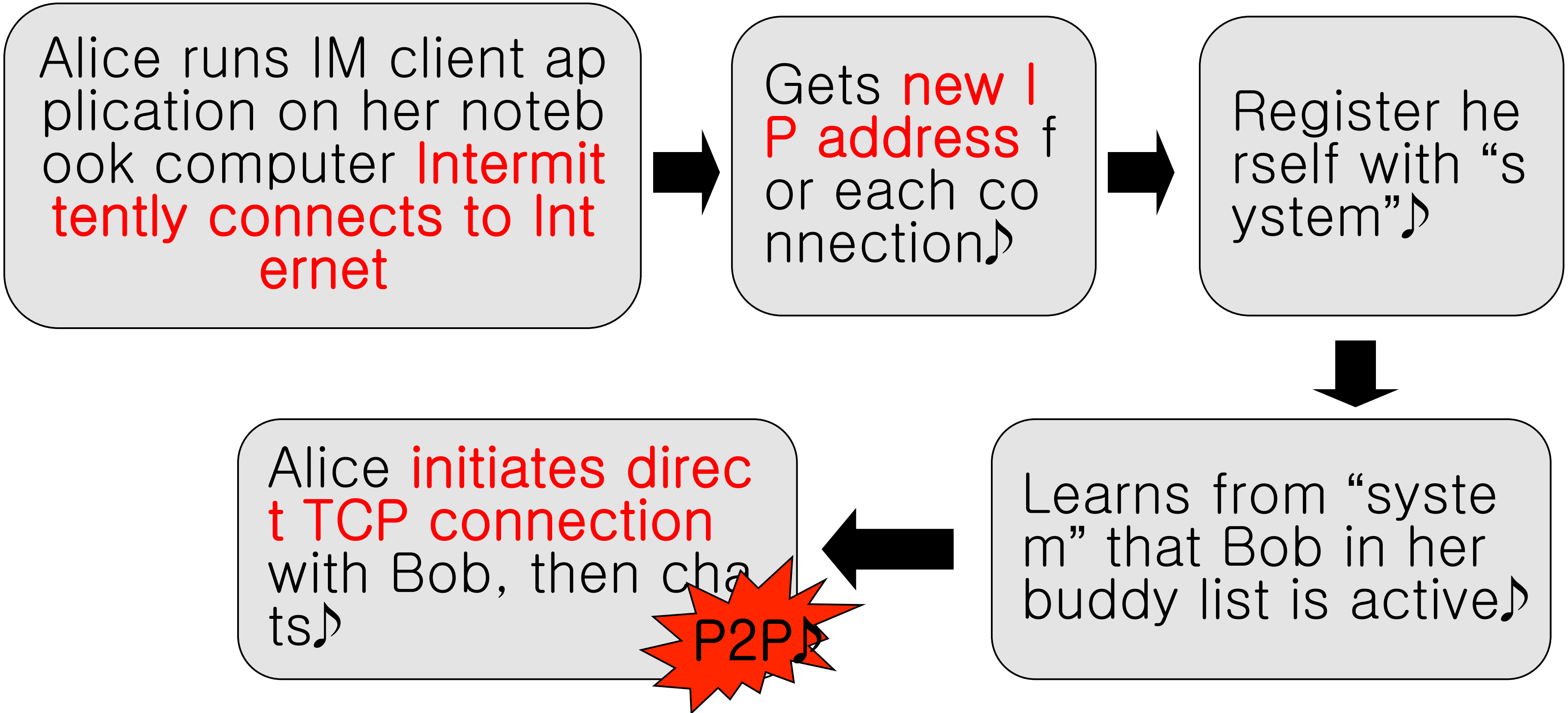
While Alice downloads, othe
r users upload from Alice.

Promising properties of P2P

- Massive scalability
- Autonomy : non single point of failure
- Resilience to Denial of Service
- Load distribution
- Resistance to censorship

P2P Communication

- Instant Messaging
- Skype is a VoIP P2P system



Alice runs IM client application on her notebook computer Intermittently connects to Internet

Gets new IP address for each connection♪

Register herself with “system”♪

Learns from “system” that Bob in her buddy list is active♪

Alice initiates direct TCP connection with Bob, then chats♪

P2P♪

P2P/Grid Distributed Processing

- seti@home
  - Search for ET intelligence
  - Central site collects radio telescope data
  - Data is divided into work chunks of 300 Kbytes
  - User obtains client, which runs in background
  - Peer sets up TCP connection to central computer, downloads chunk
  - Peer does FFT on chunk, uploads results, gets new chunk
- Not P2P communication, but exploit Peer computing power

Key Issues

- Management
  - How to maintain the P2P system under high rate of churn efficiently
- Lookup
  - How to find out the appropriate content/resource that a user wants
- Throughput
  - How to copy content fast, efficiently, reliably

Management Issue

- A P2P network must be self-organizing.
  - Join and leave operations must be self-managed.
  - The infrastructure is untrusted and the components are unreliable.
  - The number of faulty nodes grows linearly with system size.
  - Tolerance to failures and churn
    - Content replication, multiple paths
    - Leverage knowledge of executing application
- Load balancing
- Dealing with freeriders
  - Freerider : rational or selfish users who consume more than their fair share of a public resource, or shoulder less than a fair share of the costs of its production.

Lookup Issue

- How do you locate data/files/objects in a large P2P system built around a dynamic set of nodes in a scalable manner without any centralized server or hierarchy?

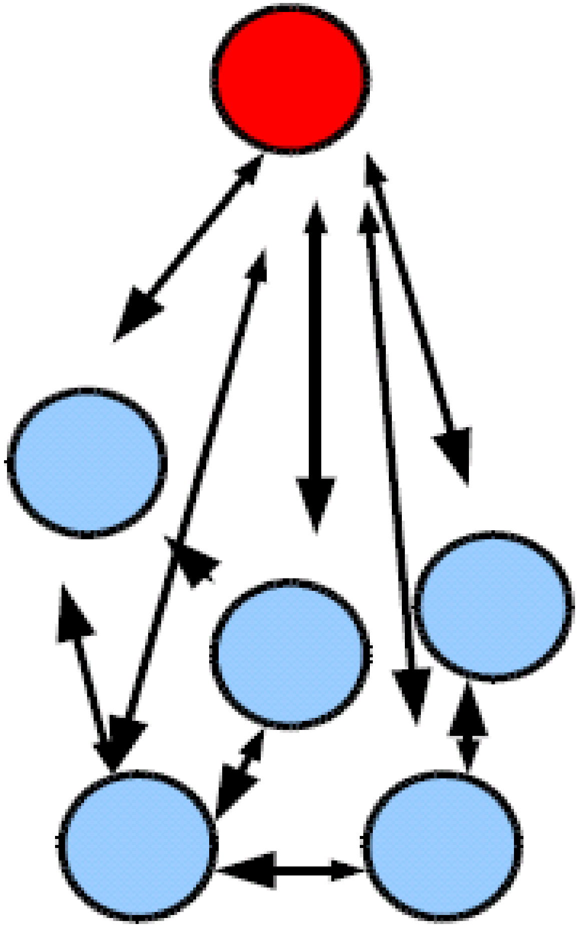
- Efficient routing even if the structure of the network is unpredictable.

Overlay Graph

- **Unstructured overlays**
  - new node randomly chooses existing nodes as neighbors
  - E.g., Napster, Gnutella, Kazaa
- **Structured overlays**
  - edges arranged in restrictive structure such as a ring
  - E.g, Chord, CAN, Pastry/Tapestry, Kademlia

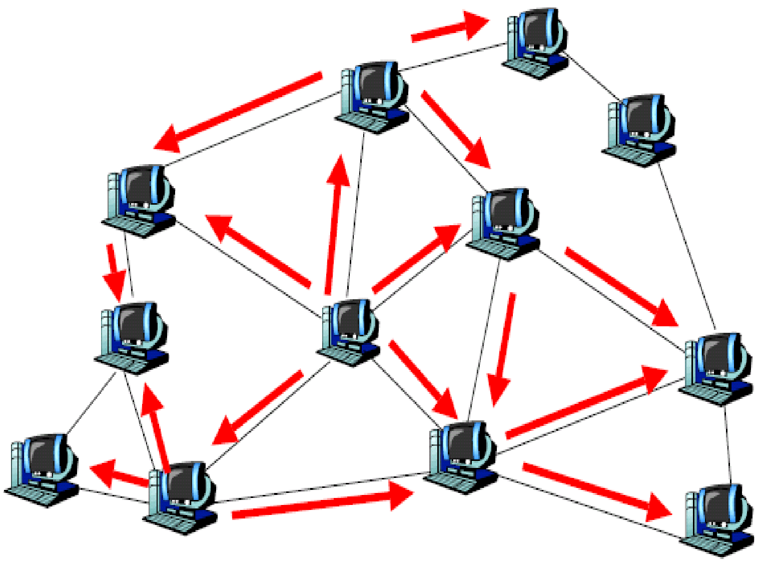
Napster

- Centralized Lookup
  - Centralized directory services
  - Steps
    - Connect to Napster server.
    - Upload list of files to server.
    - Give server keywords to search the full list with.
    - Select "best" of correct answers. (ping)
  - Performance Bottleneck
- Lookup is centralized, but files are copied in P2P manner



Gnutella

- The main representative of “unstructured P2P”
- Fully decentralized lookup for files
- Flooding based lookup
  - Peer asks neighbors (usually 7)
  - If neighbor does not have matching content, it forwards request to its neighbor (usually 7)
  - Process continues for TTL (# of hops a packet can live before it dies, usually 10).
- Inefficient lookup in terms of scalability and bandwidth



Unstructured vs Structured

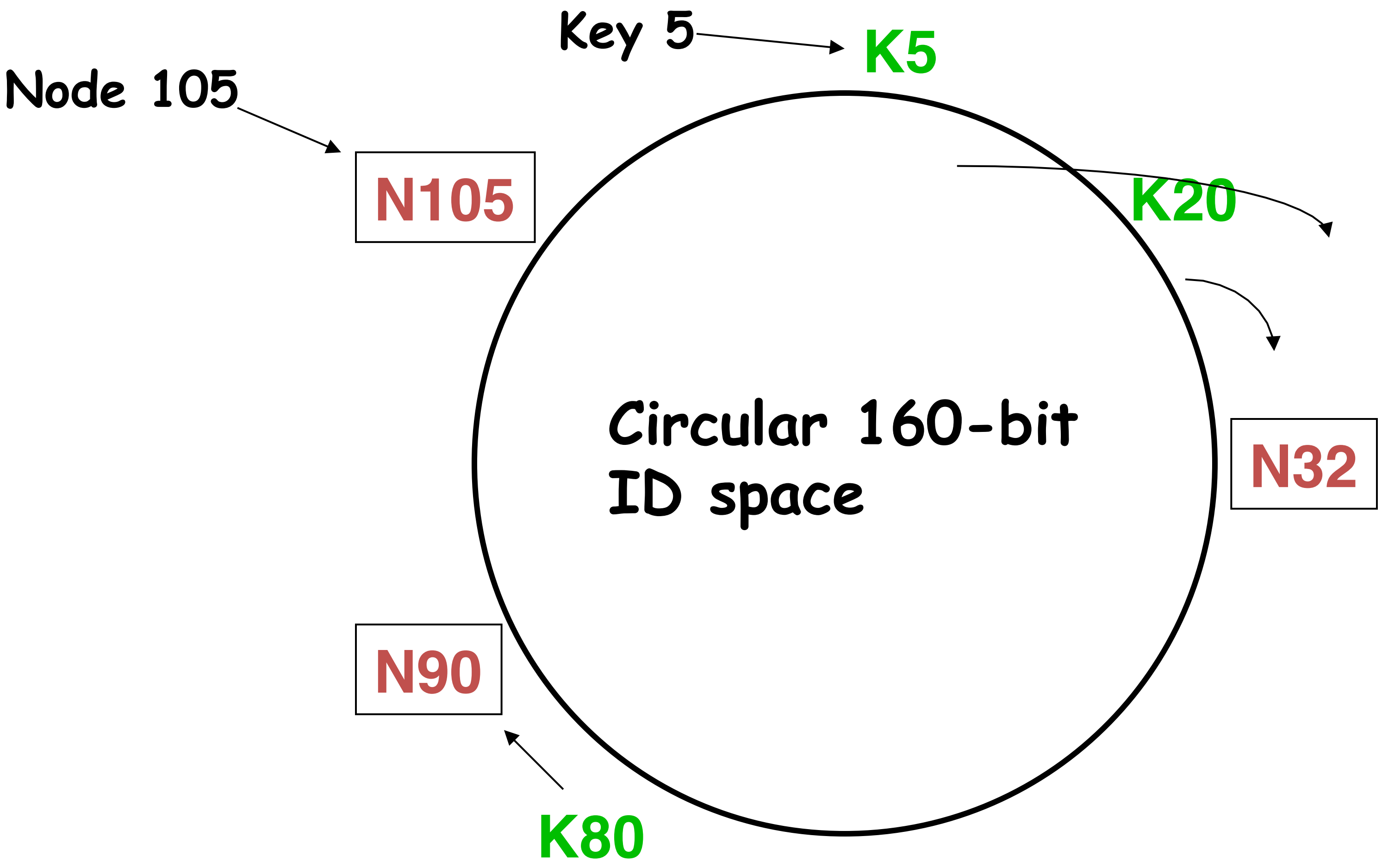
- Unstructured P2P networks allow resources to be placed at any node. The network topology is arbitrary, and the growth is spontaneous.

- Structured P2P networks simplify resource location and load balancing by defining a topology and defining rules for resource placement (e.g., in a ring, or a d-dimensional cartesian space)
  - Guarantee efficient search for rare objects

Structured P2P Systems

- Chord
  - Consistent hashing based ring structure
- Pastry
  - Uses ID space concept similar to Chord
  - Exploits concept of a nested group
- CAN
  - Nodes/objects are mapped into a d-dimensional Cartesian space
- Kademlia
  - Similar structure to Pastry, but the method to check the closeness is XOR function

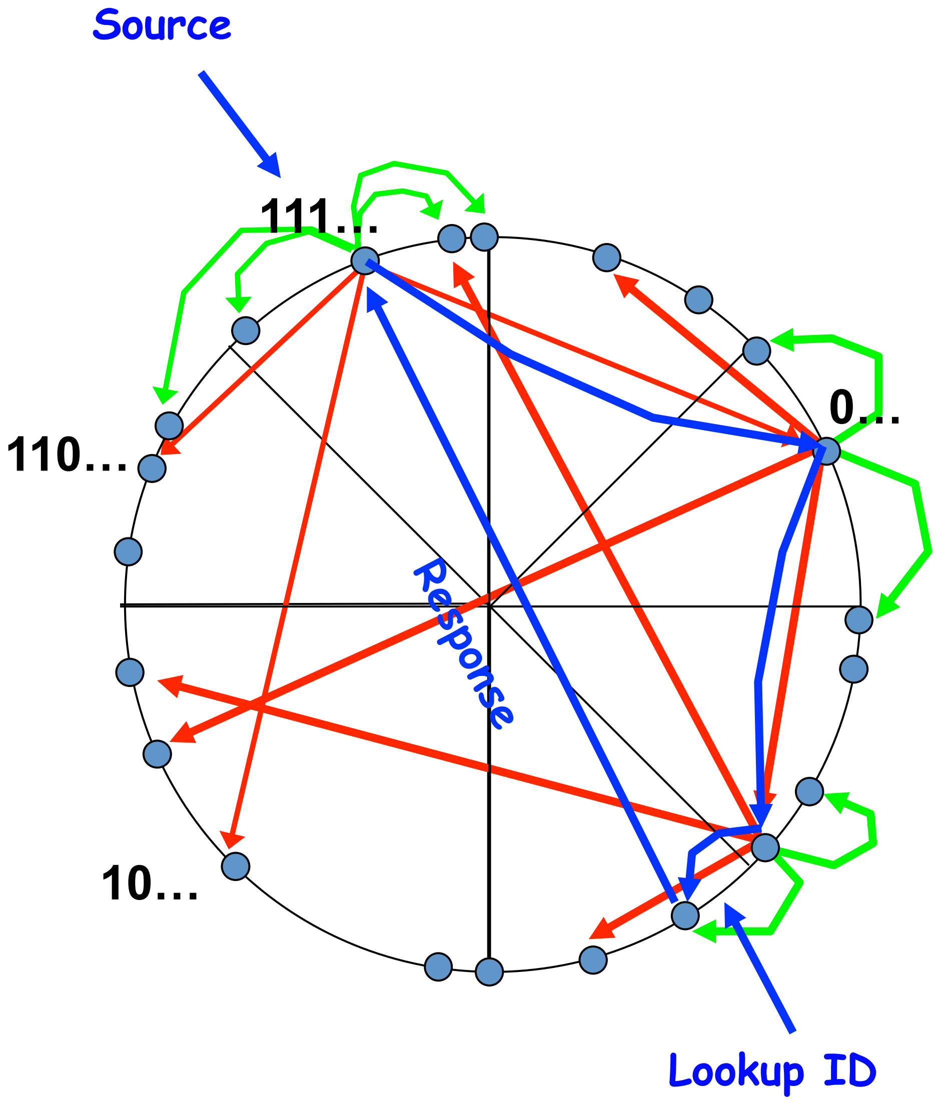
Consistent hashing



A key is stored at its successor: node with next higher ID

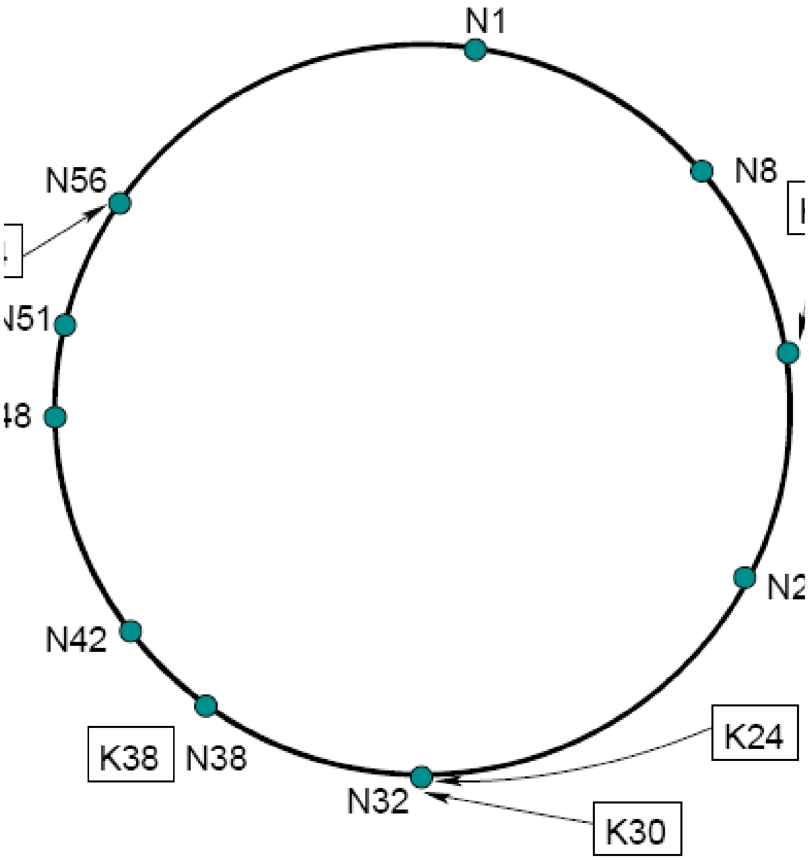
Lookup with Leaf Set

- Assign IDs to nodes
  - Map hash values to node with closest ID
- Leaf set is successors and predecessors
  - All that's needed for correctness
- Routing table matches successively longer prefixes
  - Allows efficient lookups
- Data Replication:
  - On leaf set



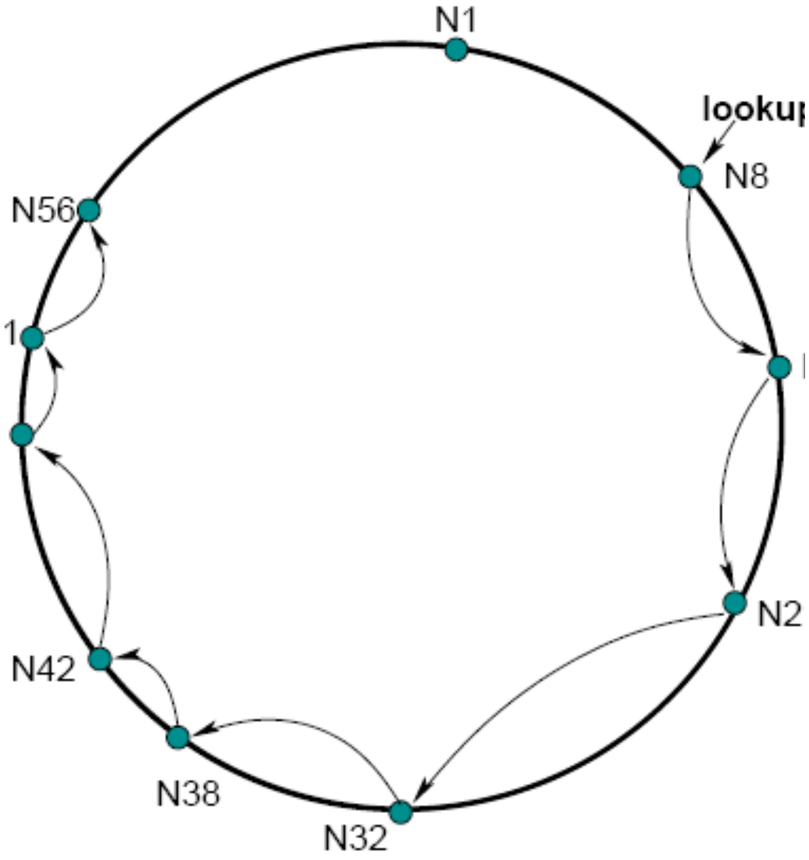
Chord

- Consistent hashing based on an ordered ring overlay
- Both keys and nodes are hashed to 160 bit IDs (SHA-1)
- Then keys are assigned to nodes using consistent hashing
  - – Successor in ID space

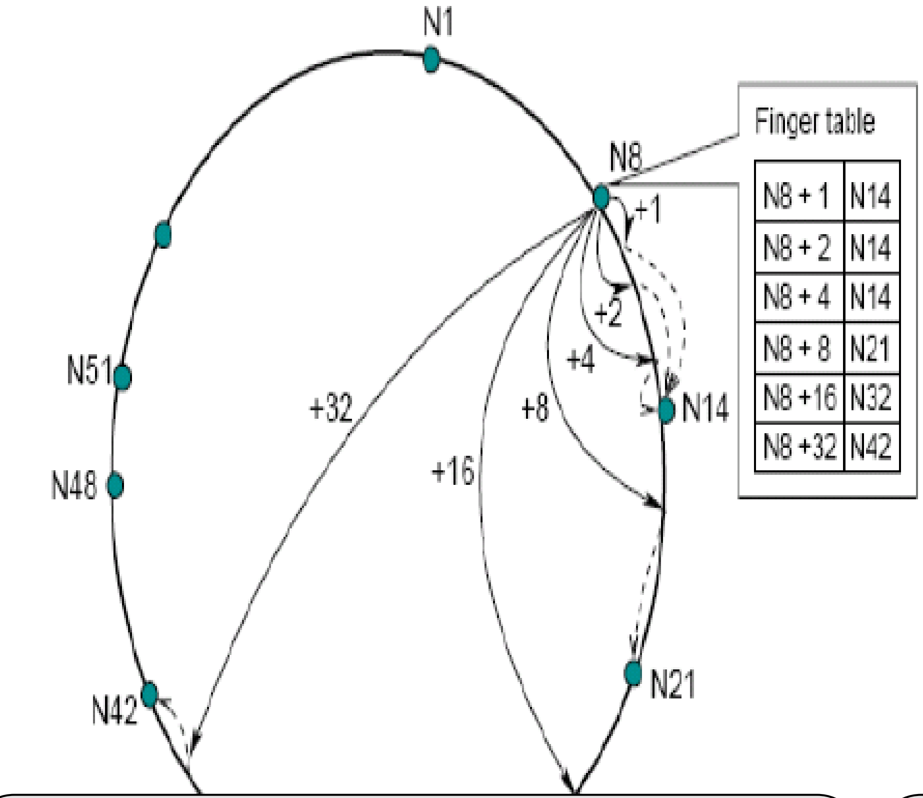


Chord : Primitive Lookup

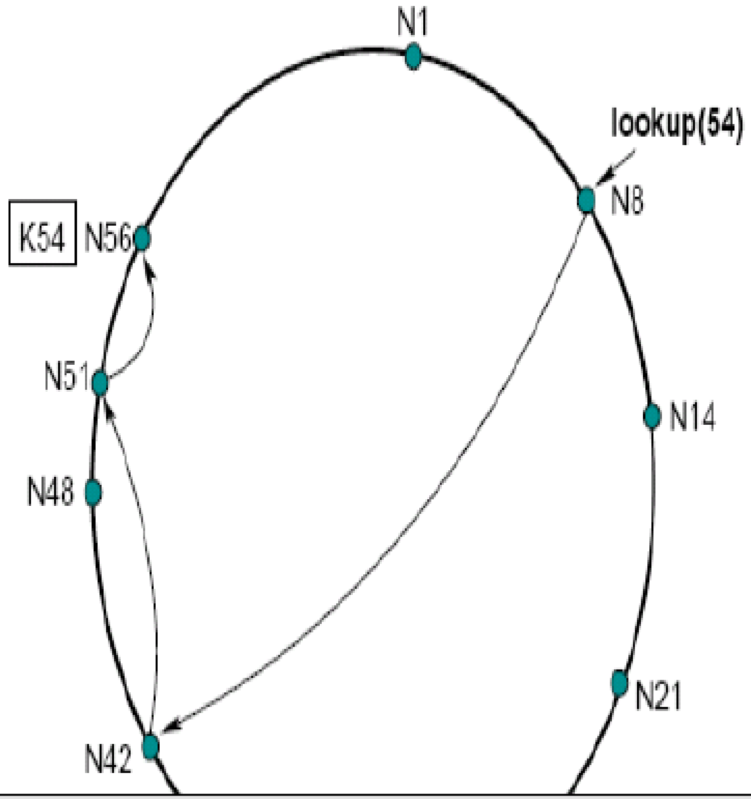
- Lookup query is forwarded to successor.
  - one way
- Forward the query around the circle
- In the worst case, O(N) forwarding is required
  - In two ways, O(N/2)



Chord : Scalable Lookup



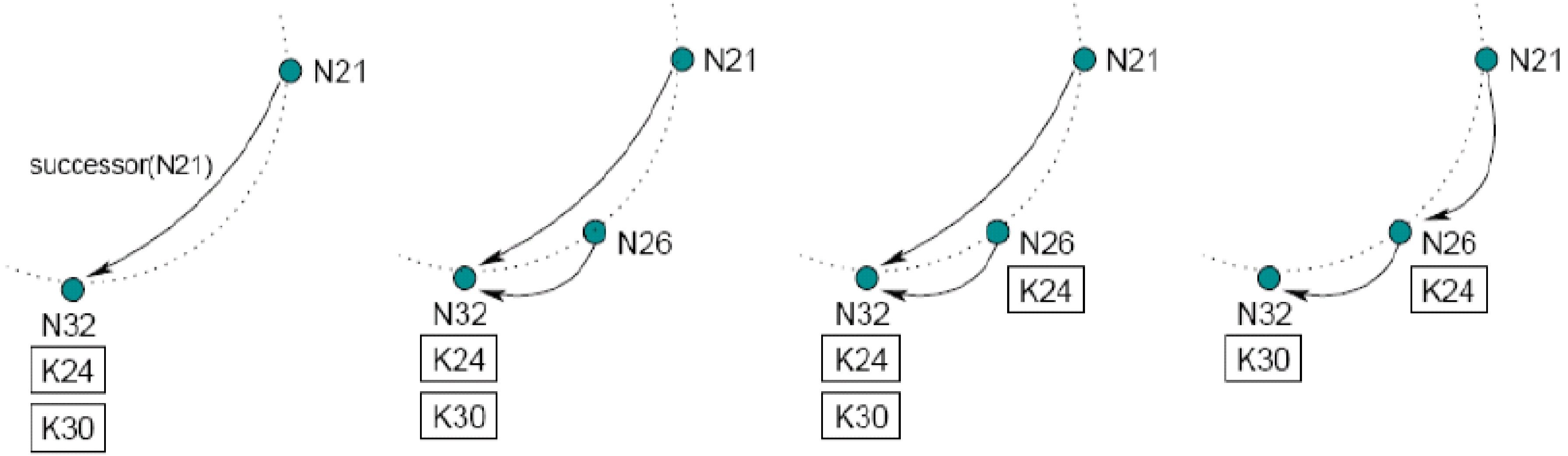
N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 +16	N32
N8 +32	N42



i_{th} entry of a finger table points the successor of the key (node ID + 2^i)

A finger table has O(log N) entries and the scalable lookup is bounde d to O(log N)

Chord : Node join



- A new node has to
  - Fill its own successor, predecessor and fingers
  - Notify other nodes for which it can be a successor, predecessor of finger
- Simpler way : Find its successor, then stabilize
  - Immediately join the ring (lookup works), then modify the structure

Chord : Failure handling

- Failed nodes are handled by
  - Replication: instead of one successor, we keep r successors
    - More robust to node failure (we can find our new successor if the old one failed)
  - Alternate paths while routing
    - If a finger does not respond, take the previous finger, or the replicas, if close enough
- At the DHT level, we can replicate keys on the r successor nodes
  - The stored data becomes equally more robust

Chord : Stabilization

- If the ring is correct, then routing is correct, fingers are needed for the speed only
- Stabilization
  - Each node periodically runs the stabilization routine
  - Each node refreshes all fingers by periodically calling find_successor(n+2i-1) for a random i
  - Periodic cost is O(logN) per node due to finger refresh

Advantages/Disadvantages of Consistent Hashing

- Advantages:
  - Automatically adapts data partitioning as node membership changes
  - Node given random key value automatically "knows" how to participate in routing and data management
  - Random key assignment gives approximation to load balance
- Disadvantages
  - Uneven distribution of key storage natural consequence of random node names ⇒ Leads to uneven query load
  - Key management can be expensive when nodes transiently fail
    - Assuming that we immediately respond to node failure, must transfer state to new node set
    - Then when node returns, must transfer state back
    - Can be a significant cost if transient failure common
- Disadvantages of "Scalable" routing algorithms
  - More than one hop to find data ⇒ O(log N) or worse
  - Number of hops unpredictable and almost always > 1
    - Node failure, randomness, etc

Why Peer to Peer Storage?

- Incremental Scalability
  - Add or remove nodes as necessary
    - Systems stays online during changes
  - With many other systems:
    - Must add large groups of nodes at once
    - System downtime during change in active set of nodes
- Low Management Overhead (related to first property)
  - System automatically adapts as nodes die or are added
  - Data automatically migrated to avoid failure or take advantage of new nodes
- Self Load-Balance
  - Automatic partitioning of data among available nodes
  - Automatic rearrangement of information or query loads to avoid hot-spots
- Not bound by commercial notions of semantics
  - Can use weaker consistency when desired
  - Can provide flexibility to vary semantics on a per-application basis
  - Leads to higher efficiency or performance

Recap…

- Cloud storage and data management layer requires support for:
  - Extremely low latency of access, On demand scalability to very large datasets and users, Very high availability, Consistency guarantees to make application programming easier, Ability to tolerate failures
- No single system or technology provides all of the above
  - Fundamental limitations due to the CAP theorem?
- Transactions offer a powerful model for consistent, failure-resilient, durable execution of applications – however, in general, may incur increased latency and non-availability.
- P2P storage offers building block for elastic storage. P2P systems are highly scalable, have low maintenance, and gracefully tolerate changes to the number of nodes.
- We will next explore a variety of cloud solutions that benefit from (or build on top of) the various distributed computing principles, design of P2P systems, and database systems.

Properties we might want of Cloud Data Platforms

- Consistency: Updates in an agreed order
- Durability: Once accepted, won't be forgotten
- Real-time responsiveness: Replies with bounded delay
- Security: Only permits authorized actions by authenticated parties
- Privacy: Won't disclose personal data
- availability: Failures can't prevent the system from providing desired services
- Coordination: actions won't interfere with one-another
- Elasticity: the storage/data platform can rapidly ramp up or down based on needs