CSE 5306
Distributed Systems

Fault Tolerance

Jia Rao

http://ranger.uta.edu/~jrao/

Failure in Distributed Systems

- Partial failure
  - Happens when one component of a distributed system fails
  - Often leaves other components unaffected
- A failure in non-distributed systems often leads to the failure of entire system
- Fault tolerance
  - The system can automatically recover from partial failures without seriously affecting the overall performance
  - i.e., the system continues to operate in an acceptable way and tolerate faults while repairs are being made

Basic Concepts

- Being fault tolerant is strongly related to
  - ✓ Dependable systems
- Dependability implies the following:
  - ✓ Availability
    - A system is ready to be used immediately
  - ✓ Reliability
    - A system can run continuously without failure
  - ✓ Safety
    - When a system temporarily fails, nothing catastrophic happens
  - ✓ Maintainability
    - A failed system can be easily repaired
- Faults
  - ✓ Transient faults, intermittent faults, permanent faults

Failure Models

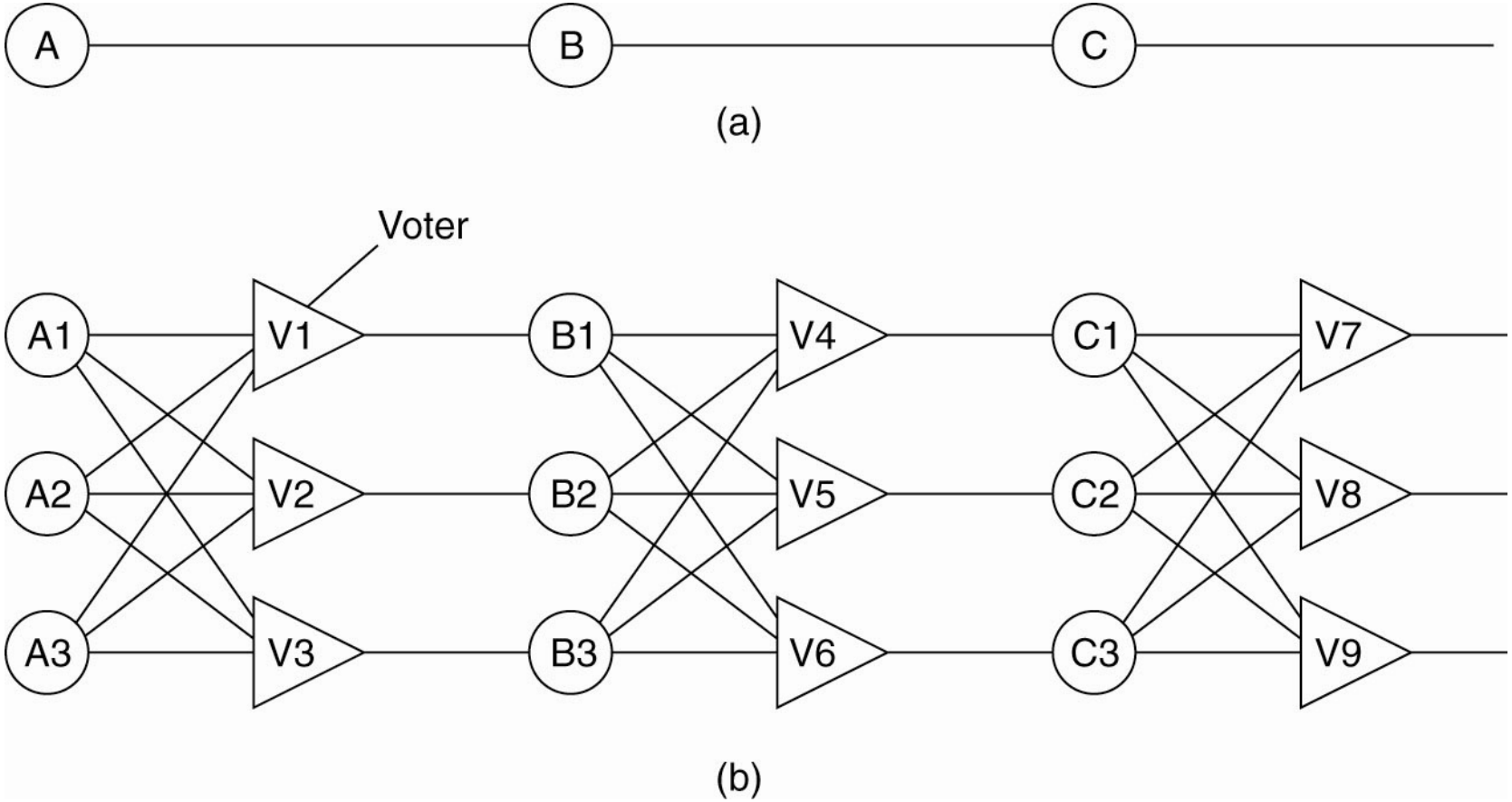
Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	A server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Different types of failures.

Failure Masking by Redundancy

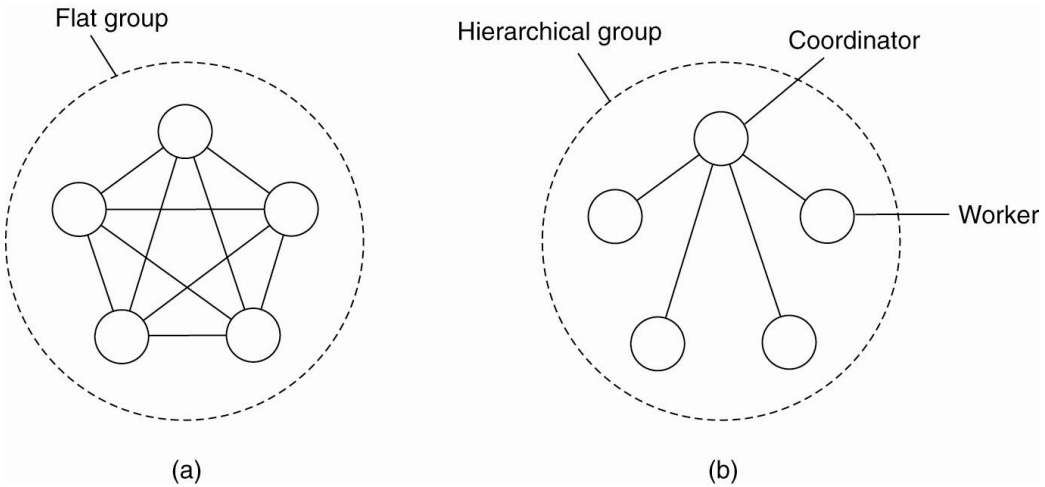
- Redundancy is the key technique for achieving fault tolerance
  - ✓ Information redundancy
    - Extra bits are added to be able to recover from errors
  - ✓ Time redundancy
    - The same action is performed multiple times to handle transient or intermittent faults
  - ✓ Physical redundancy
    - Extra equipment or processes are added to tolerate malfunctioning components

Example: Triple Modular Redundancy



Process Resilience

- Protection against process failure
  - ✓ Achieved by replicating processes into groups
  - ✓ A message to this group should be received by all members
    - Thus, if one process fails, others can take over
- Internal structure of process groups
  - ✓ Flat group v.s. hierarchical groups



Failure Masking and Replication

- A key question is: how much replication is needed to achieve fault tolerance
- A system is said to be k fault tolerant if
  - ✓ It can survive faults in k components and still meet its specification
- If the components fail silently, then having k+1 replicas is enough
- If the processes exhibit Byzantine (arbitrary) failures, a minimum of 3k+1 replicas are needed

Agreement in Faulty Systems

- The processes in a process group needs to reach an agreement in many cases
  - ✓ It is easy and straightforward when communication and processes are all perfect
  - ✓ However, when they are not, we have problems
- The goal is to have all non-faulty process reach consensus in a finite number of steps
- Different solutions may be needed, depending on:
  - ✓ Synchronous versus asynchronous systems
  - ✓ Communication delay is bounded or not
  - ✓ Message delivery is ordered or not
  - ✓ Message transmission is done through unicast or multicast

Byzantine Generals Problem (1/3)

- The original paper
  - ✓ “The Byzantine Generals Problem”, by Lamport, Shostak, Pease, In ACM Transactions on Programming Languages and Systems, July 1982
- Settings
  - ✓ Several divisions of the Byzantine army are camped outside an enemy city
    - Each division commanded by its own general
  - ✓ After observing the enemy, they must decide upon a common plan of action
  - ✓ However, some generals may be traitors
    - Trying to prevent the loyal generals from reaching agreement

Byzantine Generals Problem (2/3)

- Must guarantee that
  - ✓ All loyal generals decide upon the same plan of action
  - ✓ A small number of traitors cannot cause the loyal generals to adopt a bad plan
- A straightforward approach: simple majority voting
  - ✓ However, traitors may give different values to others
- More specifically
  - ✓ If the i_{th} general is loyal, then the value he/she sends must be used by every loyal general as the value of v(i)

Byzantine Generals Problem (3/3)

- More precisely, we have:
- A commanding general must send an order to his n-1 lieutenant generals such that
  - ✓ All loyal lieutenants obey the same order
  - ✓ If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

Impossibility Results

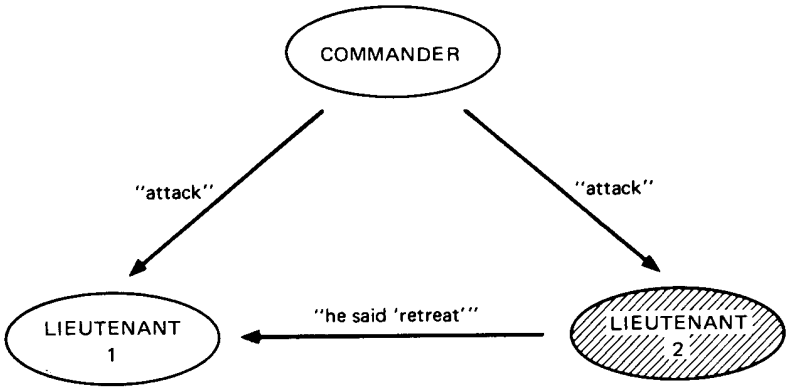


Fig. 1. Lieutenant 2 a traitor.

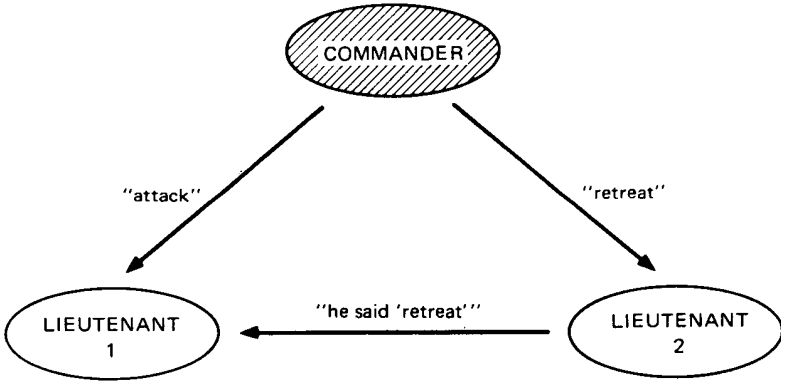
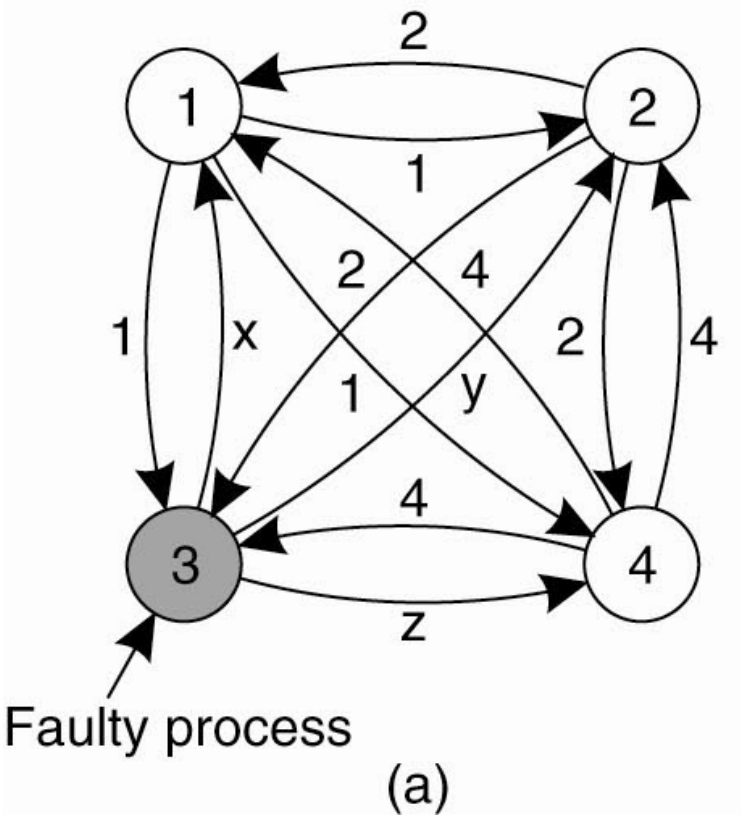


Fig. 2. The commander a traitor.

Byzantine Agreement Problem (1/3)

- The problem: reaching an agreement given
  - ✓ Three non-faulty processes
  - ✓ One faulty process
- Assume
  - ✓ Processes are synchronous
  - ✓ Messages are unicast while preserving ordering
  - ✓ Communication delay is bounded



Each process sends

their value to the others.

Byzantine Agreement Problem (2/3)

1	Got(1, 2, x, 4)
2	Got(1, 2, y, 4)
3	Got(1, 2, 3, 4)
4	Got(1, 2, z, 4)

(b)

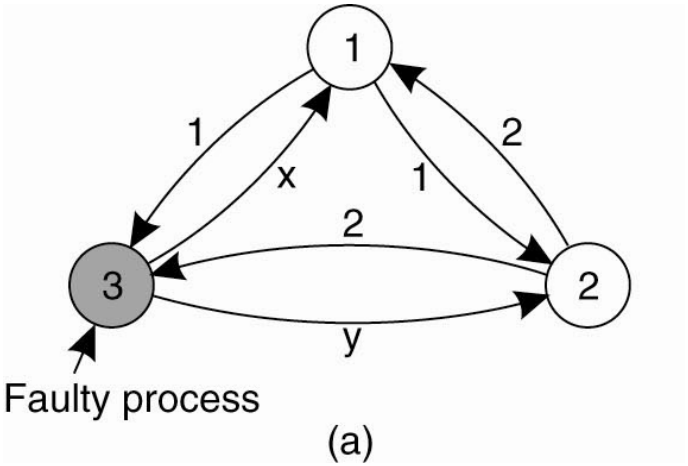
1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

The Byzantine agreement problem for three non-faulty and one faulty process. (b) The vectors that each process assembles based on (a).
(c) The vectors that each process receives in step 3.

Byzantine Agreement Problem (3/3)

- In a system with k faulty processes, an agreement can be achieved only if
  - ✓ 2k+1 correctly functioning processes are present, for a total of 3k+1 processes



(a)

1 Got(1, 2, x)
2 Got(1, 2, y)
3 Got(1, 2, 3)

(b)

1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

(c)

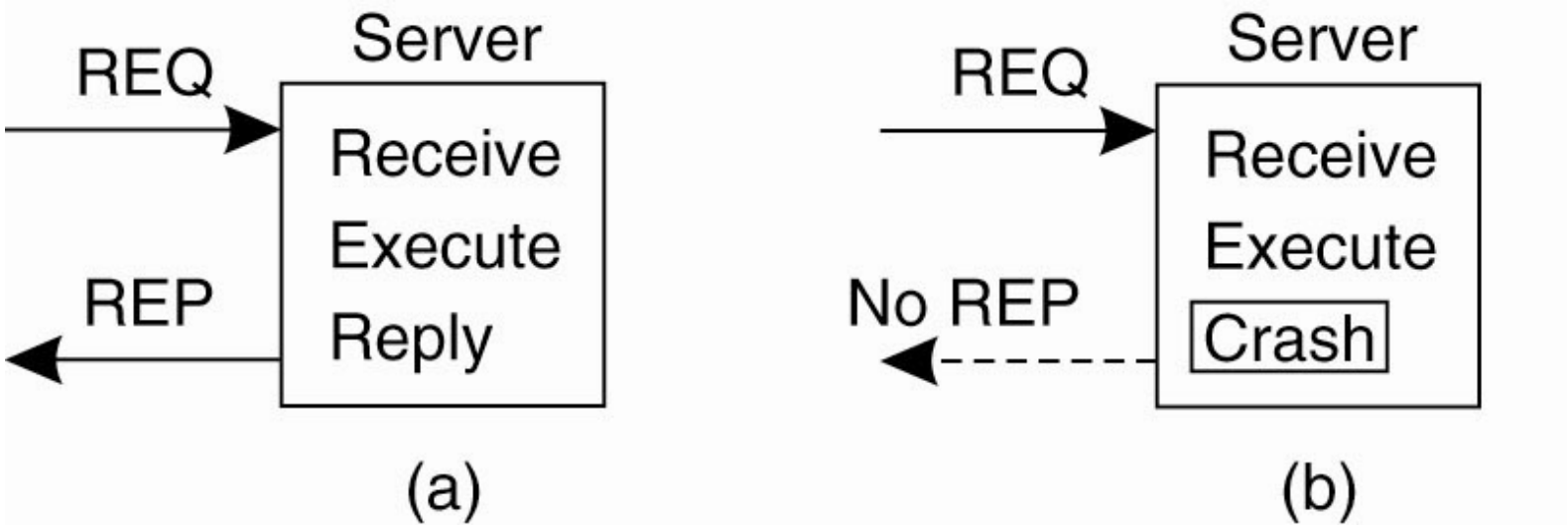
Failure Detection

- It is critical to detect faulty components
  - ✓ So that we can do proper recovery
- A common approach is to actively ping processes with a time-out mechanism
  - ✓ Faulty if no response within a given time limit
  - ✓ Can be a side-effect of regular message exchanging
- The problem with the “ping” approach
  - ✓ It is hard to determine if no response is due to node failure or just communication failure

Reliable Client-Server Communication

- In addition to process failures, another important class of failure is communication failures
- Point-to-point communication
  - ✓ Reliability can be achieved by protocols such as TCP
  - ✓ However, TCP itself may fail, and the distributed system will need to mask such TCP crash failure
- Remote procedure call (RPC): transparency is the challenge
  - ✓ The client is unable to locate the server
  - ✓ The request message from the client to the server is lost
  - ✓ The server crashes after receiving a request
  - ✓ The reply message from the server to the client is lost
  - ✓ The client crashes after send a request

Server Crash

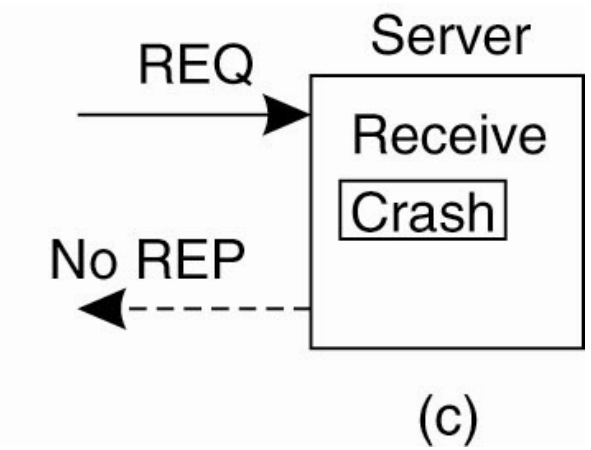


A server in client-server communication.

(a) The normal case.

(b) Crash after execution.

(c) Crash before execution.



Recovery from Server Crashes

- The challenge is that
  - ✓ A client does not know if server crashes before execution or crashes after execution
  - ✓ Two situations should be handled differently
- Three schools of thought for client OS
  - ✓ At least once semantics
  - ✓ At most once semantics
  - ✓ To guarantee nothing
- Ideally, we like exactly once semantics
  - ✓ But in general, there is no way to arrange this

Example: Printing Text (1/3)

- Assume the client
  - ✓ Request the server to print some text
  - ✓ Got ACK when the request is delivered
- Two strategies at the server
  - ✓ Send a completion message right before it tells the printer
  - ✓ Send a completion message after text has been printed
- The server crashes and then recover and announce to all clients that he is up and running again
  - ✓ The question is what the client should do
  - ✓ The client does not know if its request will be actually carried out by the server

Example: Printing Text (2/3)

- Four strategies at the client
  - ✓ Never reissue a request: text may not be printed
  - ✓ Always reissue a request: text may be printed twice
  - ✓ Reissue a request only if it did not receive the acknowledgement of its request
  - ✓ Reissue a request only if it has received the acknowledgement of its request
- Three events that could happen at the server
  - ✓ Send the completion message (M), print the text (P), and crash (C)
  - ✓ Six different orderings: MPC, MC(P), PMC, PC(M), C(PM), C(MP)

Example: Printing Text (3/3)

Client	Server					
	Strategy M → P			Strategy P → M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

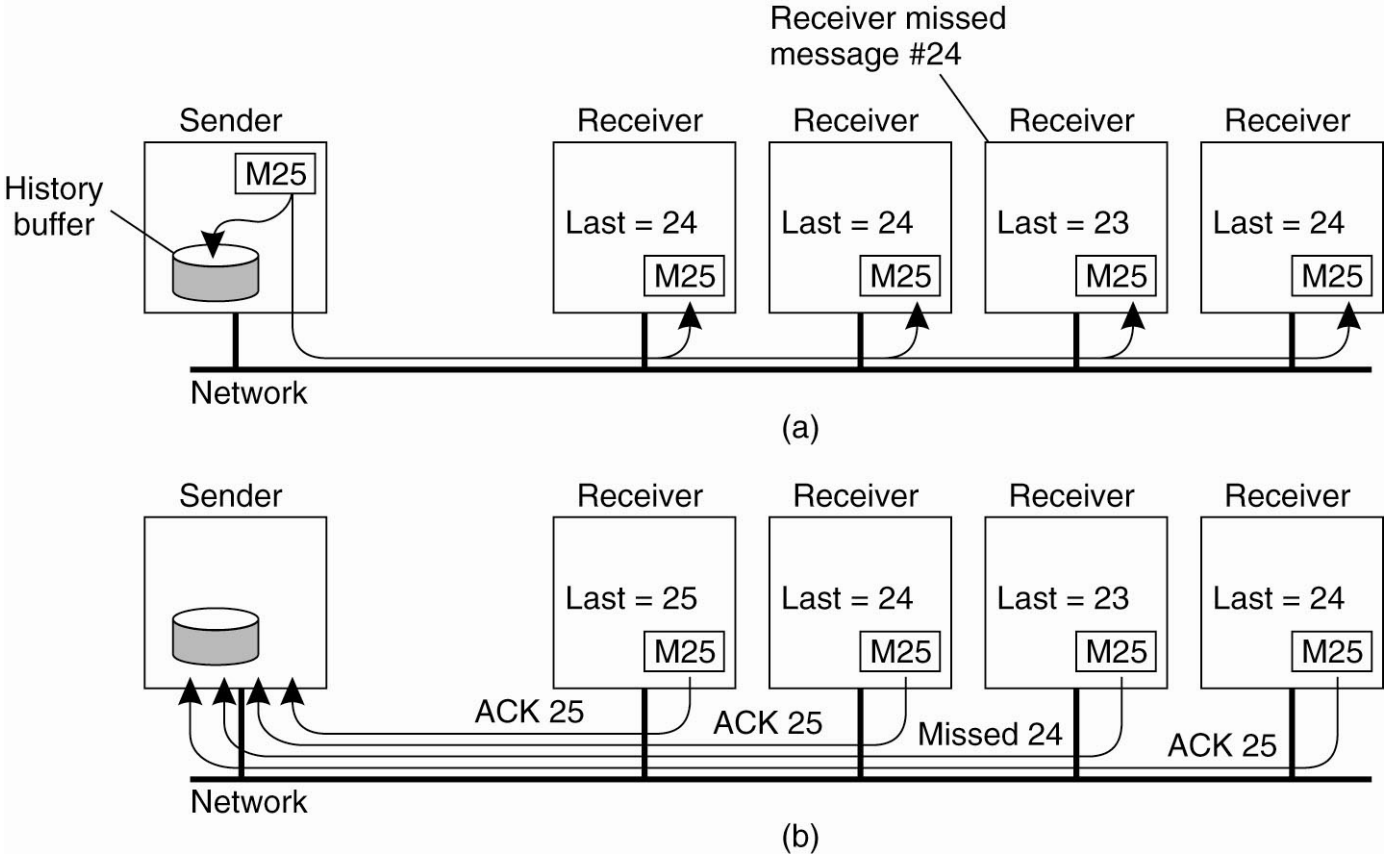
OK = Text is printed once
DUP = Text is printed twice
ZERO = Text is not printed at all

Different combinations of client and server strategies in the presence of server crashes.

Lost Reply Message

- A common solution is to set a timer
  - ✓ If the timer expires, send the request again
- However, the client cannot tell why there was no reply
  - ✓ The request gets lost in the channel? Or the server is just slow?
- If the request is idempotent, then we can always reissue a request with no harm
  - ✓ We can structure requests in an idempotent way
  - ✓ However, this is not always true, e.g., money transfer
- Other possible solutions
  - ✓ Ask the server to keep a sequence number
  - ✓ Use a bit in the message indicating if it is the original request

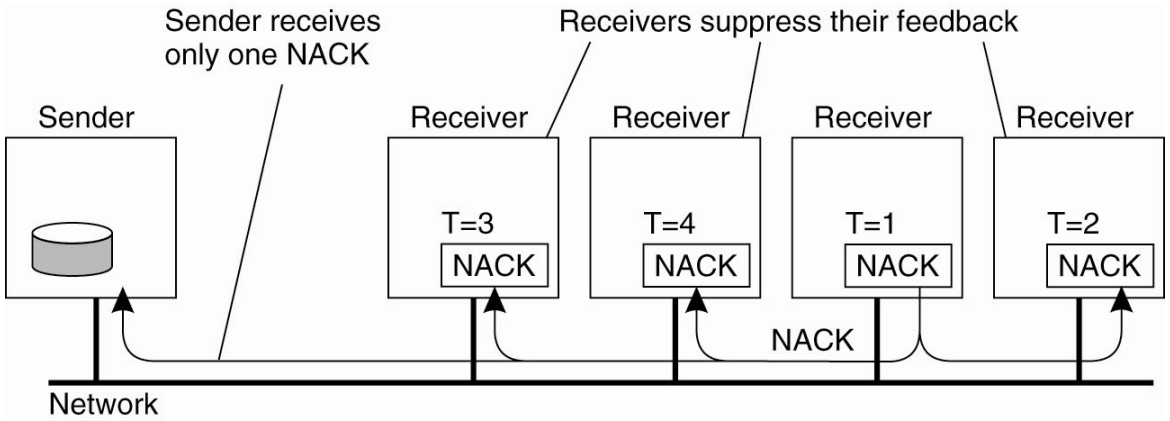
Basic Reliable-Multicasting Schemes



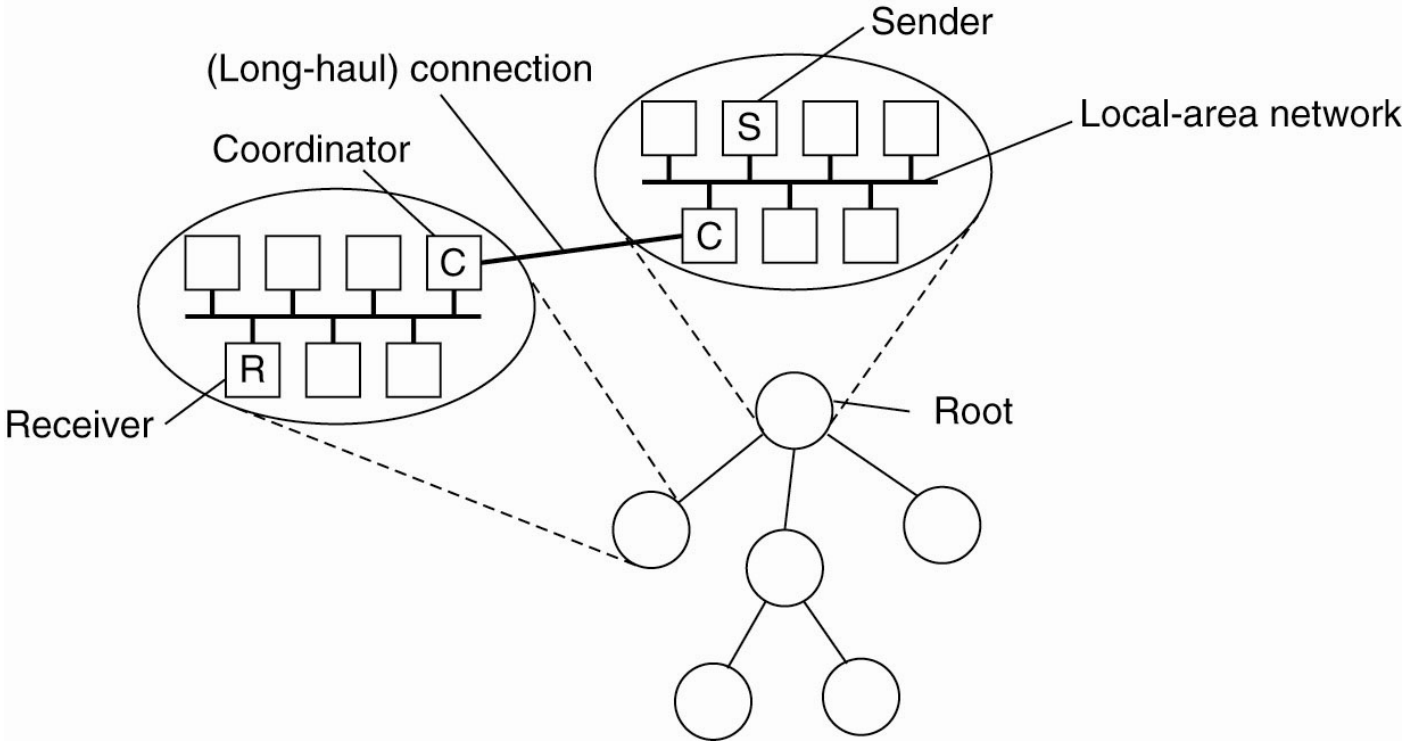
A simple solution to reliable multicasting when all receivers are known and are assumed not to fail. (a) Message transmission. (b) Reporting feedback.

Scalability in Reliable Multicasting

- The basic scheme discussed has some limitations
  - ✓ If there are N receivers, the sender must be prepared to receive N ACKs
    - Only send NACKs, but still no guarantee
  - ✓ The sender has to keep old messages
    - Set a limit on the buffer, no retransmission for very old messages
- Nonhierarchical feedback control
  - ✓ Several receivers have scheduled a request for retransmission, but the first retransmission request leads to the suppression of others



Hierarchical Feedback Control



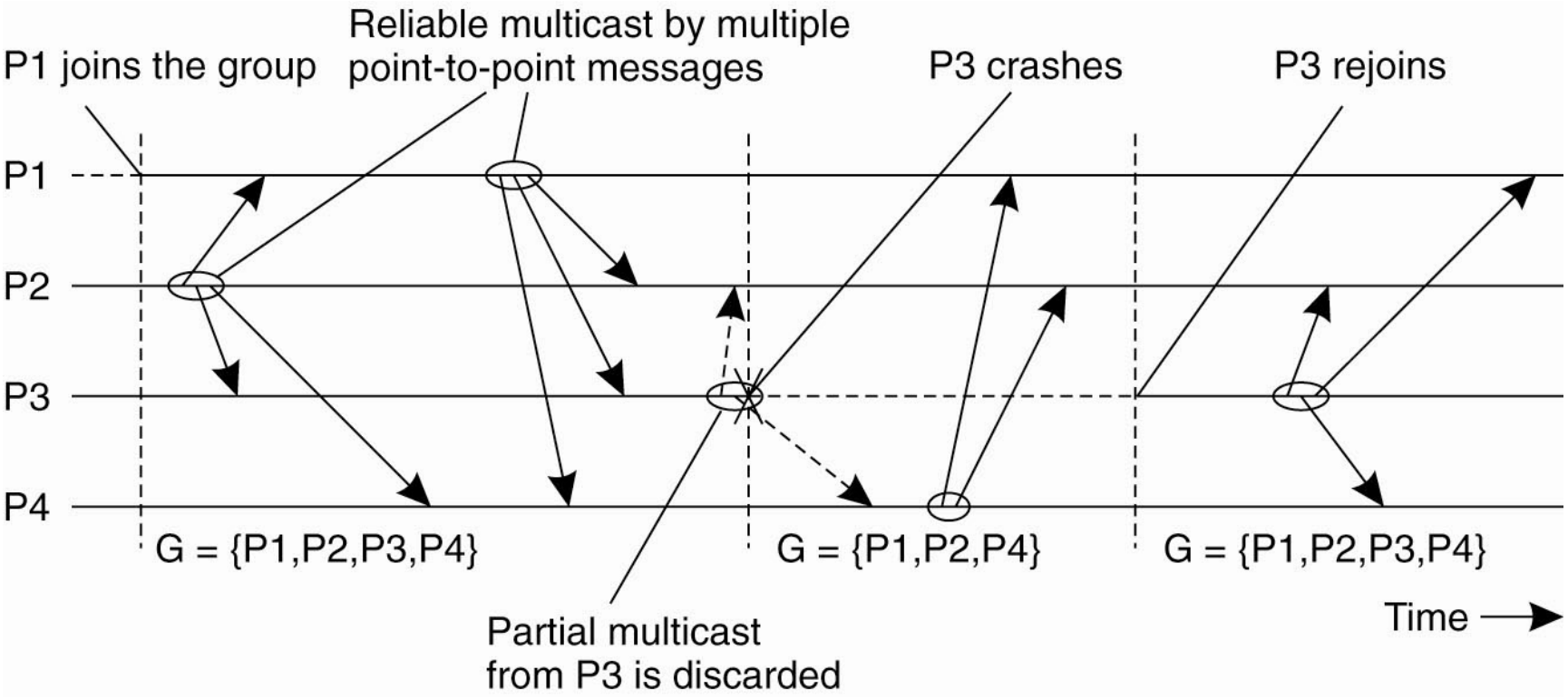
The essence of hierarchical reliable multicasting. Each local coordinator forwards the message to its children and later handles retransmission requests.

Atomic Multicast

- Consider a replicated database system constructed on top of a distributed system, we require that
  - ✓ An update should be either performed at all replicas or none at all
  - ✓ All updates should be done in the same order in all replicas
- The atomic multicast problem
  - ✓ A message is delivered to either all processes or to none
    - Virtually synchronous
  - ✓ Messages are delivered in the same order to all processes
    - Message ordering

Virtual Synchrony

- The principle of virtual synchronous multicast
  - ✓No multicast can pass the view-change barrier



Message Ordering (1/3)

- Virtual synchrony does not address the ordering of multicast
- The are four different cases
  - ✓ Unordered multicast
    - Receivers may receive messages in a different order
  - ✓ FIFO-ordered multicast
    - The messages from the same sender should be received in the same order as they are sent
  - ✓ Causally-ordered multicasts
    - If a message m1 causally precedes m2, then m1 should be always received before m2 at any receiver, even if the senders are different
  - ✓ Totally-ordered multicast
    - Messages are delivered to all receivers in the same order
    - They may not be FIFO-ordered or causally-ordered

Message Ordering (2/3)

Process P1	Process P2	Process P3
sends m1	receives m1	receives m2
sends m2	receives m2	receives m1

Three communicating processes in the same group. The ordering of events per process is shown along the vertical axis.

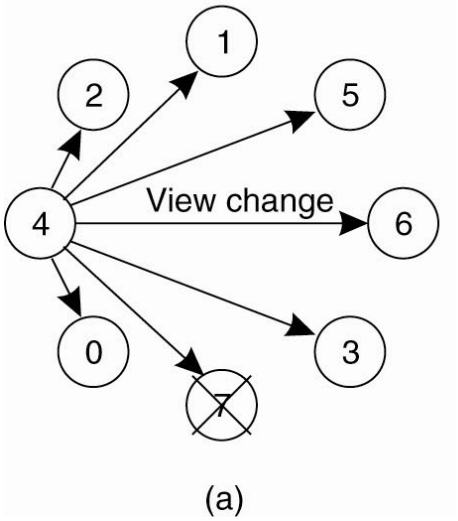
Message Ordering (3/3)

Process P1	Process P2	Process P3	Process P4
sends m1	receives m1	receives m3	sends m3
sends m2	receives m3	receives m1	sends m4
	receives m2	receives m2	
	receives m4	receives m4	

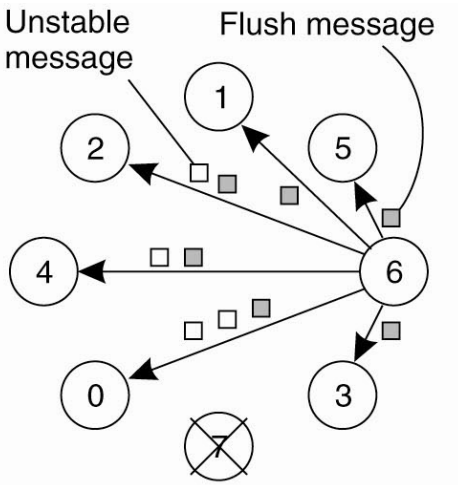
Four processes in the same group with two different senders, and a possible delivery order of messages under FIFO-ordered multicasting

Implementing Virtual Synchrony

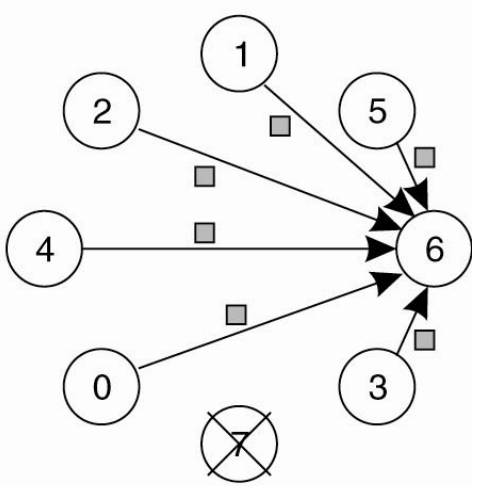
- What we will discuss is the implementation in Isis
  - ✓ A fault-tolerant distributed system that is used in industry for many years
- Assume point-to-point communication is reliable
- The task is to deliver all unstable messages before view changes
  - ✓ M is stable if one knows for sure that it has been received by all members



(a)



(b)

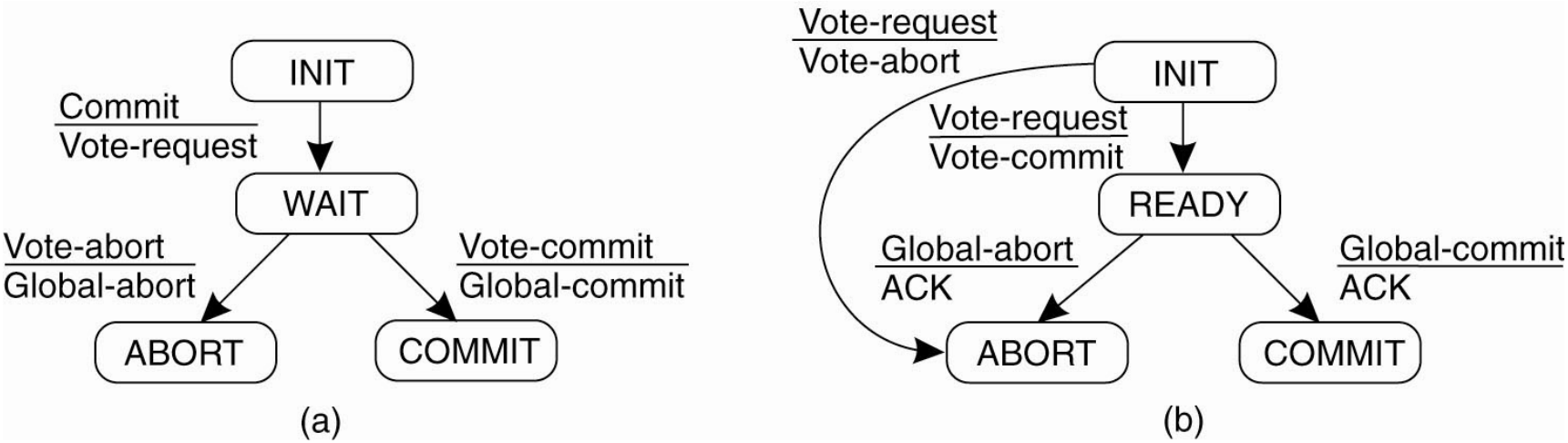


(c)

Distributed Commit

- Requires an operation being performed by all processes in the group or none at all
  - ✓ Atomic multicasting is an example of this general problem
- It is often achieved by means of a coordinator
  - ✓ One-phase commit protocol
    - The coordinator tells everyone what to do
    - No feedback when a member may fail to perform
  - ✓ Two-phase commit protocol
    - Cannot efficiently handle the failure of the coordinator
  - ✓ Three-phase commit protocol

Two-Phase Commit



(a) The finite state machine for the coordinator in 2PC.

(b) The finite state machine for a participant.

Handling Failures

- Both coordinator or participants may fail
  - ✓ Timeout mechanisms are often applied, and
  - ✓ Each saves its state to persistent storage
- If a participant is in INIT state
  - ✓ Abort if no request from coordinator within a given time limit
- If the coordinator is in WAIT state
  - ✓ Abort if not all votes are collected within a given time limit
- If a participant is in READY state
  - ✓ We cannot simply decide to abort since
    - A GLOBAL_COMMIT or GLOBAL_ABORT may have been issued
  - ✓ Let everyone block until coordinator recovers
  - ✓ Contact other participants for more informed decision

Actions to Take in READY State

State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

Actions taken by a participant P when residing in state READY and having contacted another participant Q.

Actions by coordinator:

```
write START_2PC to local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected {
        wait for any incoming vote;
        if timeout {
                write GLOBAL_ABORT to local log;
                multicast GLOBAL_ABORT to all participants;
                exit;
        }
        record vote;
}
```

```
if all participants sent VOTE_COMMIT and coordinator votes COMMIT {
        write GLOBAL_COMMIT to local log;
        multicast GLOBAL_COMMIT to all participants;
} else {
        write GLOBAL_ABORT to local log;
        multicast GLOBAL_ABORT to all participants;
}
```

actions by participant:

```
write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
        write VOTE_ABORT to local log;
        exit;
}
if participant votes COMMIT {
        write VOTE_COMMIT to local log;
        send VOTE_COMMIT to coordinator;
        wait for DECISION from coordinator;
        if timeout {
                multicast DECISION_REQUEST to other participants;
                wait until DECISION is received; /* remain blocked */
                write DECISION to local log;
        }
        if DECISION == GLOBAL_COMMIT
                write GLOBAL_COMMIT to local log;
        else if DECISION == GLOBAL_ABORT
                write GLOBAL_ABORT to local log;
} else {
        write VOTE_ABORT to local log;
        send VOTE_ABORT to coordinator;
}
```

(a)

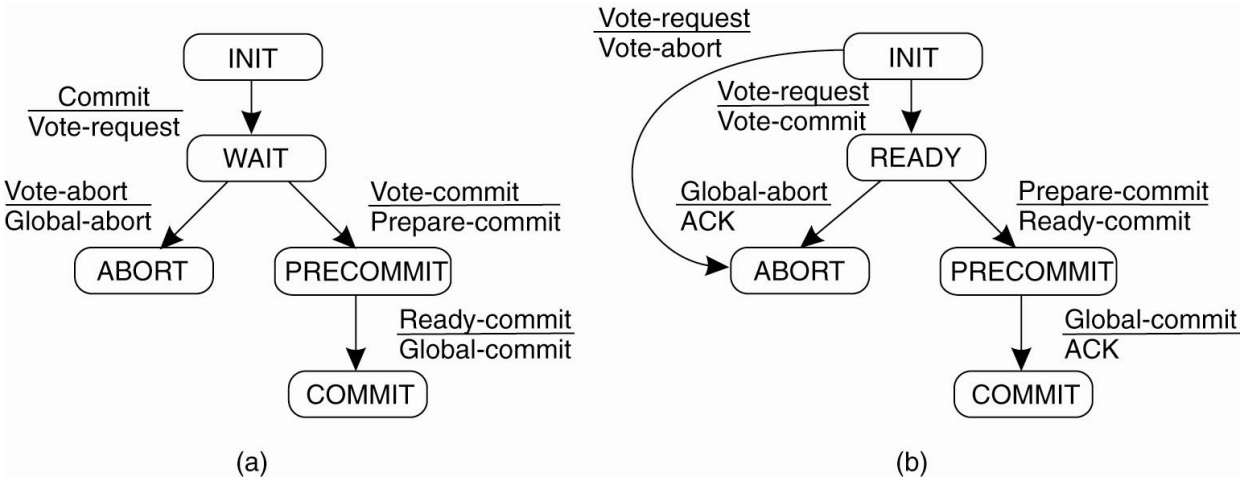
Actions for handling decision requests: /* executed by separate thread */

```
while true {
    wait until any incoming DECISION_REQUEST is received; /* remain blocked */
    read most recently recorded STATE from the local log;
    if STATE == GLOBAL_COMMIT
        send GLOBAL_COMMIT to requesting participant;
    else if STATE == INIT or STATE == GLOBAL_ABORT
        send GLOBAL_ABORT to requesting participant;
    else
        skip; /* participant remains blocked */
}
```

(b)

Three-Phase Commit

- Two-phase commit is a blocking commit protocol
  - ✓ When all participants are in READY state, no decision can be made until coordinator recovers



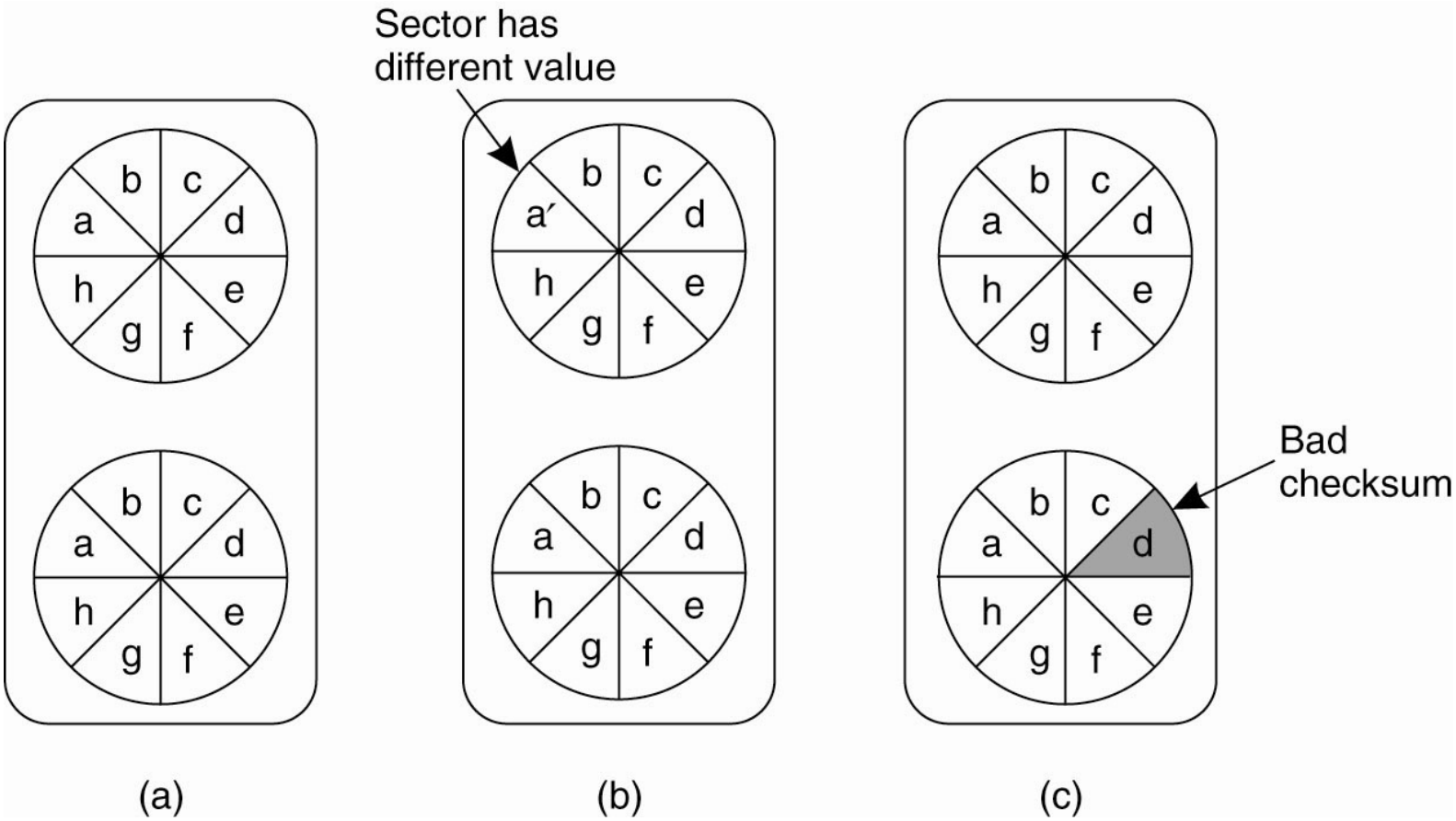
(a) The finite state machine for the coordinator in 3PC.

(b) The finite state machine for a participant.

Error Recovery

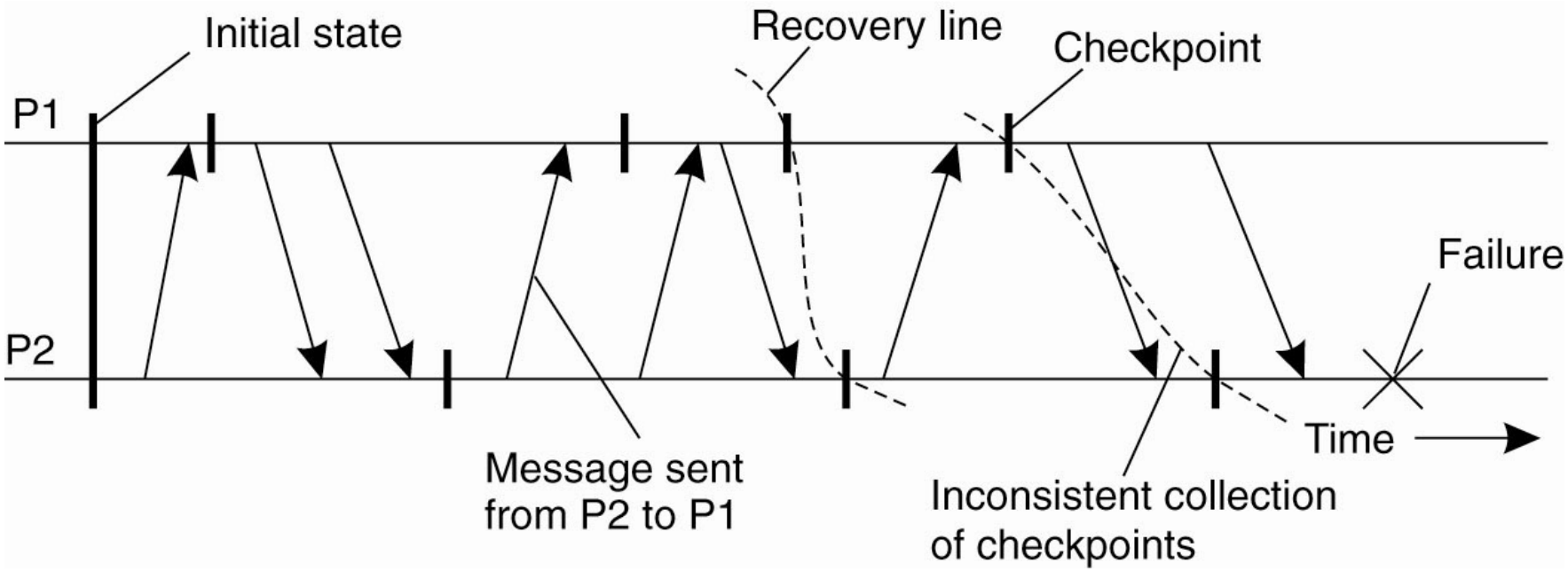
- Two forms of error recovery
  - ✓ **Backward recovery** brings the system from its present erroneous state back to a previously correct state, e.g., checkpointing
  - ✓ **Forward recovery** brings the system to a correct new state from which it can continue to execute, e.g., erasure code

Recovery – Stable Storage



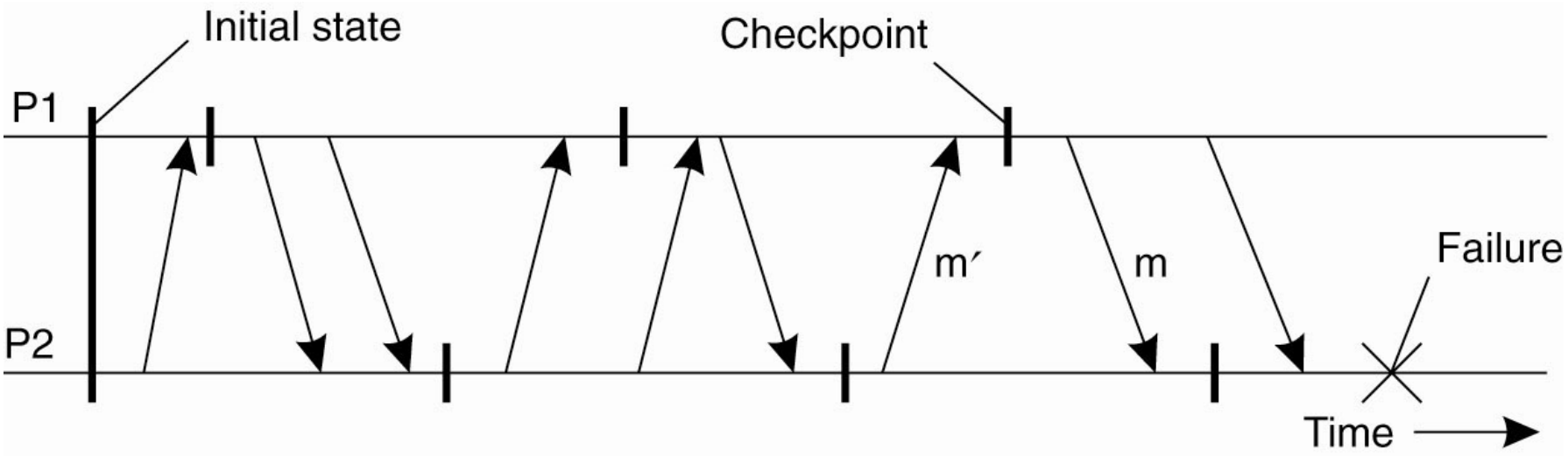
(a) Stable storage. (b) Crash after drive 1 is updated. (c) Bad spot.

Checkpointing



A recovery line.

Independent Checkpointing



The domino effect.

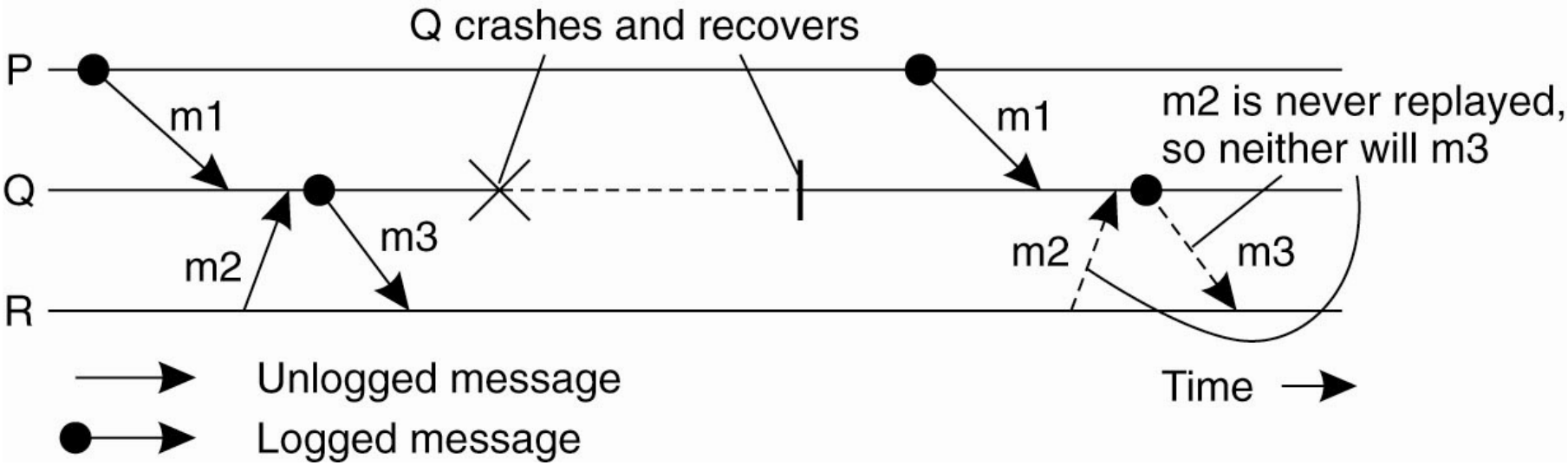
Coordinated Checkpointing

- Synchronize the checkpointing in all processes
  - ✓ The saved state is automatically globally consistent
- Achieved by using a two-phase blocking protocol
  - ✓ The coordinator multicasts a request to do checkpoint
  - ✓ Upon receiving such a request, a process queues any subsequent message and notify the coordinator that it has taken a checkpoint
  - ✓ When the coordinator receives all notifications, it multicasts a CHECKPOINT_DONE message
  - ✓ Everyone moves forward after seeing CHECKPOINT_DONE

Message Logging

- Checkpointing is expensive,
  - ✓ It is thus important to reduce the number of checkpointing
- The main intuition is
  - ✓ If we can replay all the transmission since the last checkpoint, we can reach a globally consistent state
  - ✓ i.e., trade off communication with frequent checkpointing
- The challenge of message logging is how to deal with orphan process
  - ✓ i.e., the process survived the crash, but is in an inconsistent state with the crashed process after recovery

Orphan Process – An Example



Incorrect replay of messages
after recovery, leading to an orphan process.

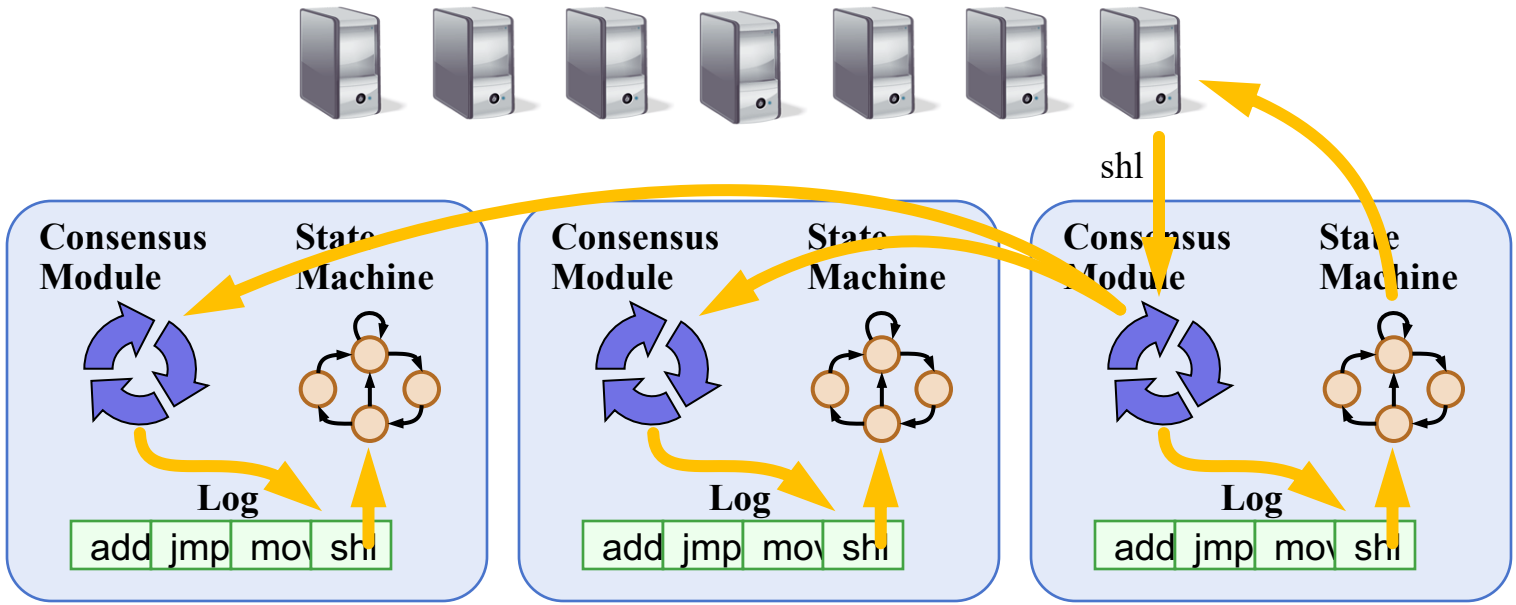
Orphan Process - Definition

- A message m is said to be stable if
  - ✓ It can no longer be lost, e.g., it has been written to stable storage
- DEP(m): include processes that depend on the delivery of m
  - ✓ i.e., the processes to which m has been delivered
  - ✓ If m' causally depends on m, then DEP(m') $\subset$ DEP(m)
- COPY(m): include processes that have a copy of m, but m has not been written to stable storage
  - ✓ If all these processes crashes, we can never replay m
- Orphan process Q can then be precisely defined as
  - ✓ There exists m such that Q $\in$ DEP(m) but everyone in COPY(m) has crashed, i.e., it depends on m but m can no longer be replayed

Handling Orphan Process

- Our objective is
  - ✓ The ensure that if process in COPY(m) crashes, then no surviving process left in DEP(m), i.e., DEP(m)$\subset$COPY(m)
- Thus, whenever a process becomes dependent on m, it should keep a copy of m
  - ✓ This is hard since it may be too late when you realize that you are dependent on m
- Pessimistic logging protocols: ensures that
  - ✓ Each non-stable message is delivered to at most one process, i.e., there is at most one process dependent on a non-stable message
- Optimistic logging protocols
  - ✓ Any orphan process is rolled back so that it is not in DEP(m)

Goal: Replicated Log



- Replicated log => replicated state machine
  - ✓ All servers execute same commands in same order
- Consensus module ensures proper log replication
- System makes progress as long as any majority of servers are up
- Failure model: fail-stop (not Byzantine), delayed/lost messages

Slides adapted from Ousterhout and Ongaro@Stanford

The Paxos Approach

Decompose the problem:

- Basic Paxos ("single decree"):
  - ✓ One or more servers propose values
  - ✓ System must agree on a single value as chosen
  - ✓ Only one value is ever chosen
- Multi-Paxos:
  - ✓ Combine several instances of Basic Paxos to agree on a series of values forming the log

Slides adapted from Ousterhout and Ongaro@Stanford

Requirements for Basic Paxos

- Safety:
  - ✓ Only a single value may be chosen
  - ✓ A server never learns that a value has been chosen unless it really has been
- Liveness (as long as majority of servers up and communicating with reasonable timeliness):
  - ✓ Some proposed value is eventually chosen
  - ✓ If a value is chosen, servers eventually learn about it

The term “consensus problem” typically refers to this single-value formulation

Paxos Components

- Proposers:
  - ✓ Active: put forth particular values to be chosen
  - ✓ Handle client requests
- Acceptors:
  - ✓ Passive: respond to messages from proposers
  - ✓ Responses represent votes that form consensus
  - ✓ Store chosen value, state of the decision process
  - ✓ Want to know which value was chosen

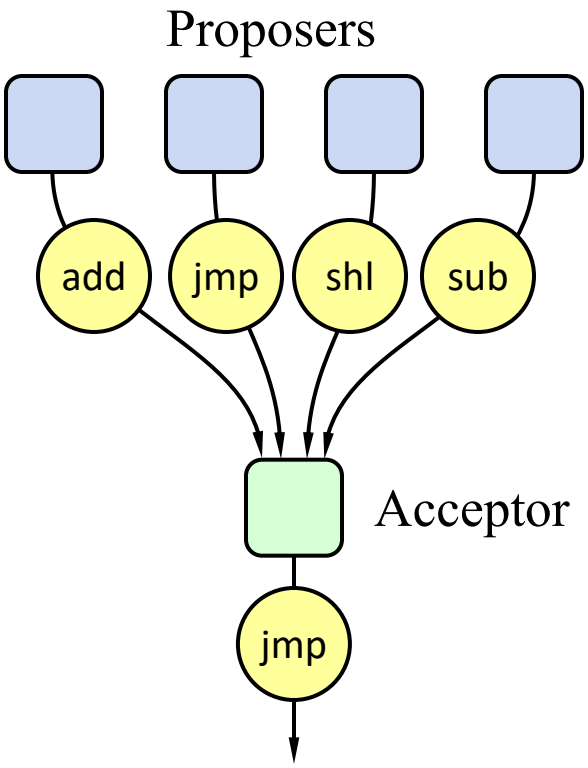
For this presentation:

- ✓ Each Paxos server contains both components

Slides adapted from Ousterhout and Ongaro@Stanford

Single Acceptor

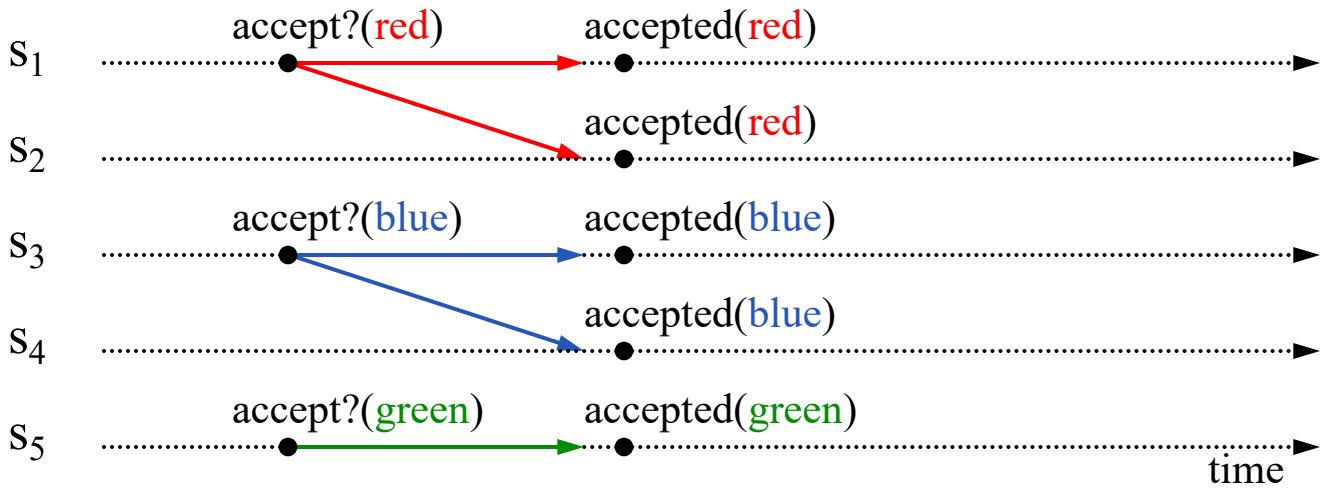
- Simple (incorrect) approach: a single acceptor chooses value
- What if acceptor crashes after choosing?
- Solution: quorum
  - ✓ Multiple acceptors (3, 5, ...)
  - ✓ Value v is chosen if accepted by majority of acceptors
  - ✓ If one acceptor crashes, chosen value still available



Slides adapted from Ousterhout and Ongaro@Stanford

Problem: Split Votes

- Acceptor accepts only first value it receives?
- If simultaneous proposals, no value might be chosen

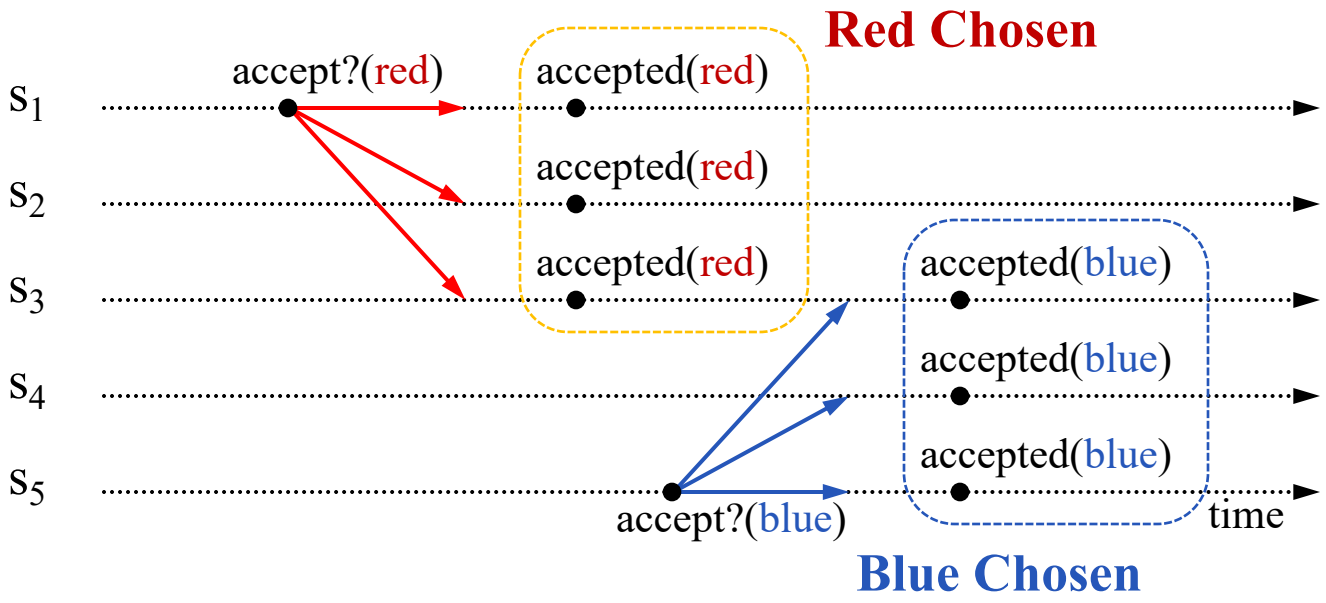


Acceptors must sometimes accept multiple (different) values

Slides adapted from Ousterhout and Ongaro@Stanford

Problem: Conflicting Choices

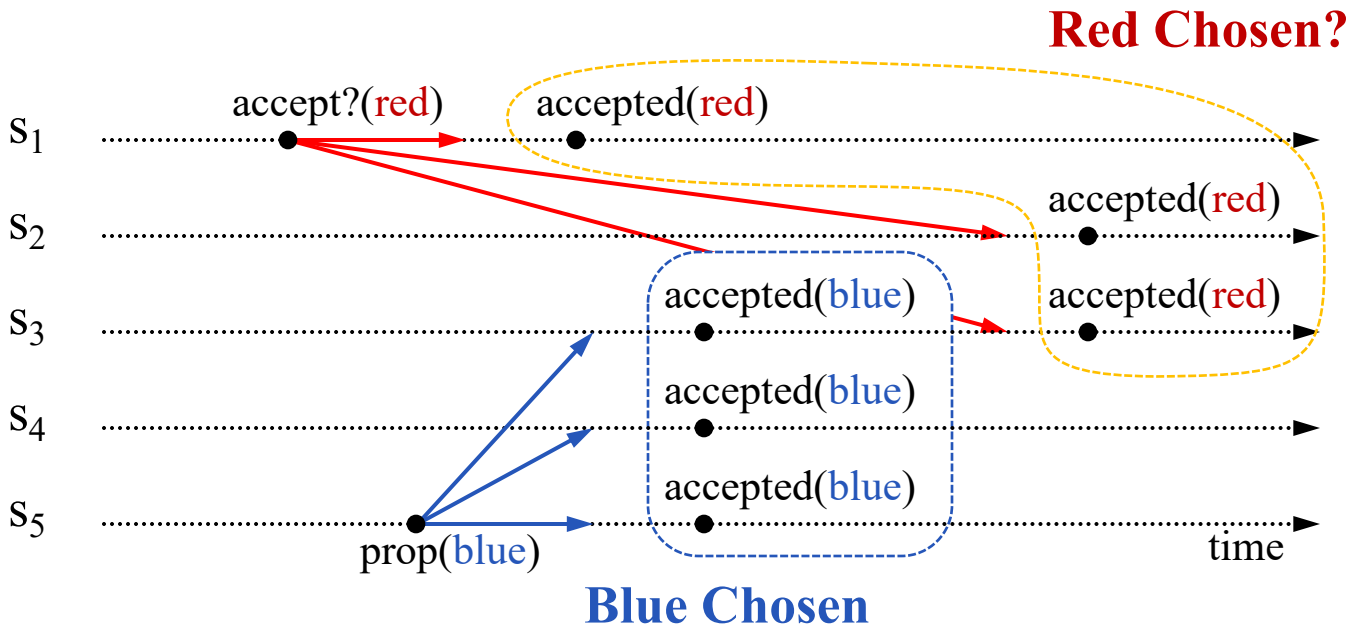
- Acceptor accepts every value it receives?
- Could choose multiple values



Once a value has been chosen, future proposals must propose/choose that same value (2-phase protocol)

Slides adapted from Ousterhout and Ongaro@Stanford

Conflicting Choices, cont'd



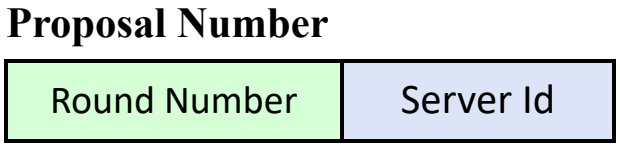
- s_5 needn't propose red (it hasn't been chosen yet)
- s_1's proposal must be aborted (s_3 must reject it)

Must order proposals, reject old ones

Slides adapted from Ousterhout and Ongaro@Stanford

Proposal Numbers

- Each proposal has a unique number
  - ✓ Higher numbers take priority over lower numbers
  - ✓ It must be possible for a proposer to choose a new proposal number higher than anything it has seen/used before
- One simple approach:



  - ✓ Each server stores maxRound: the largest Round Number it has seen so far
  - ✓ To generate a new proposal number:
    - Increment maxRound
    - Concatenate with Server Id
  - ✓ Proposers must persist maxRound on disk: must not reuse proposal numbers after crash/restart

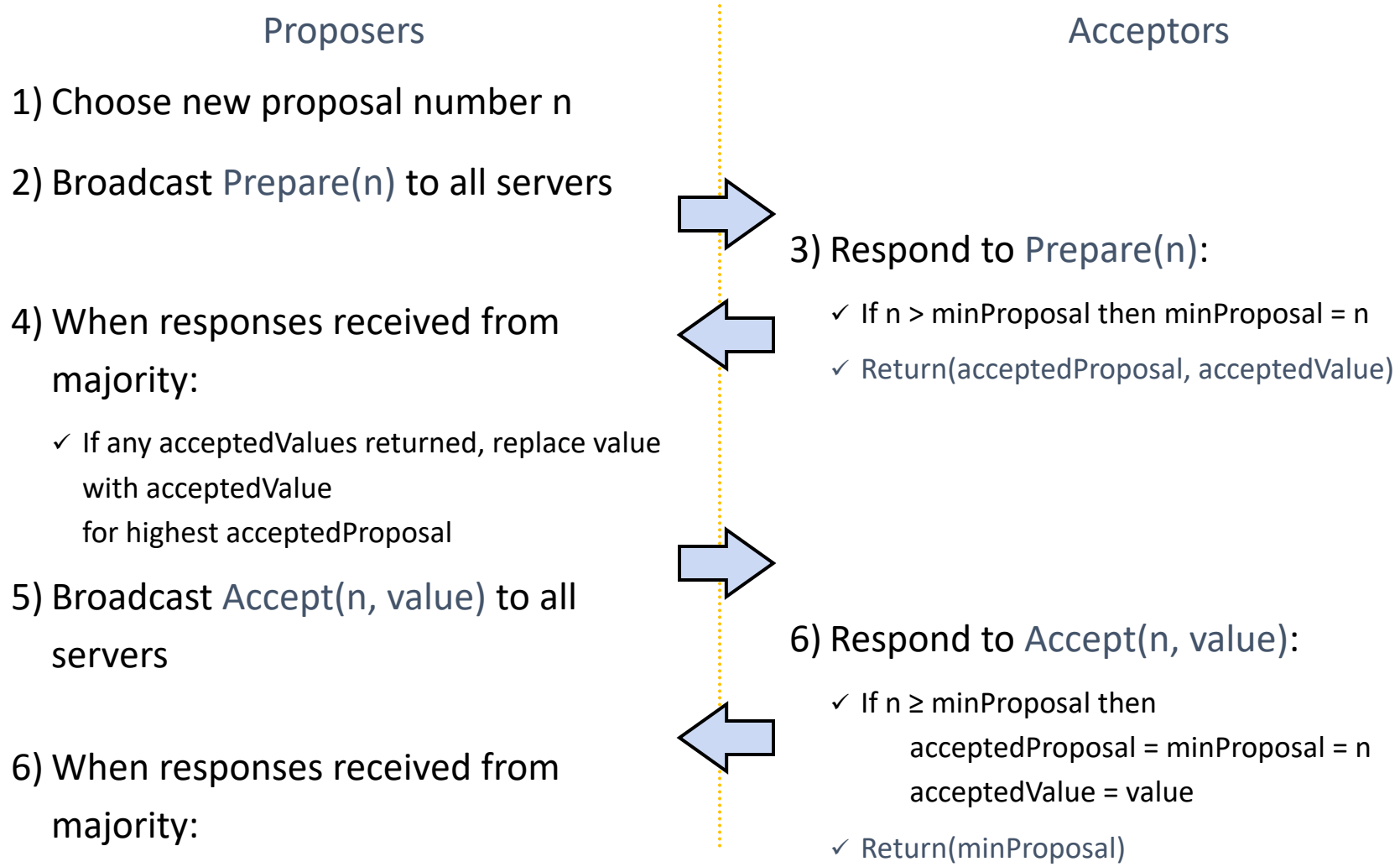
Basic Paxos

Two-phase approach:

- Phase 1: broadcast Prepare RPCs
  - ✓ Find out about any chosen values
  - ✓ Block older proposals that have not yet completed
- Phase 2: broadcast Accept RPCs
  - ✓ Ask acceptors to accept a specific value

Slides adapted from Ousterhout and Ongaro@Stanford

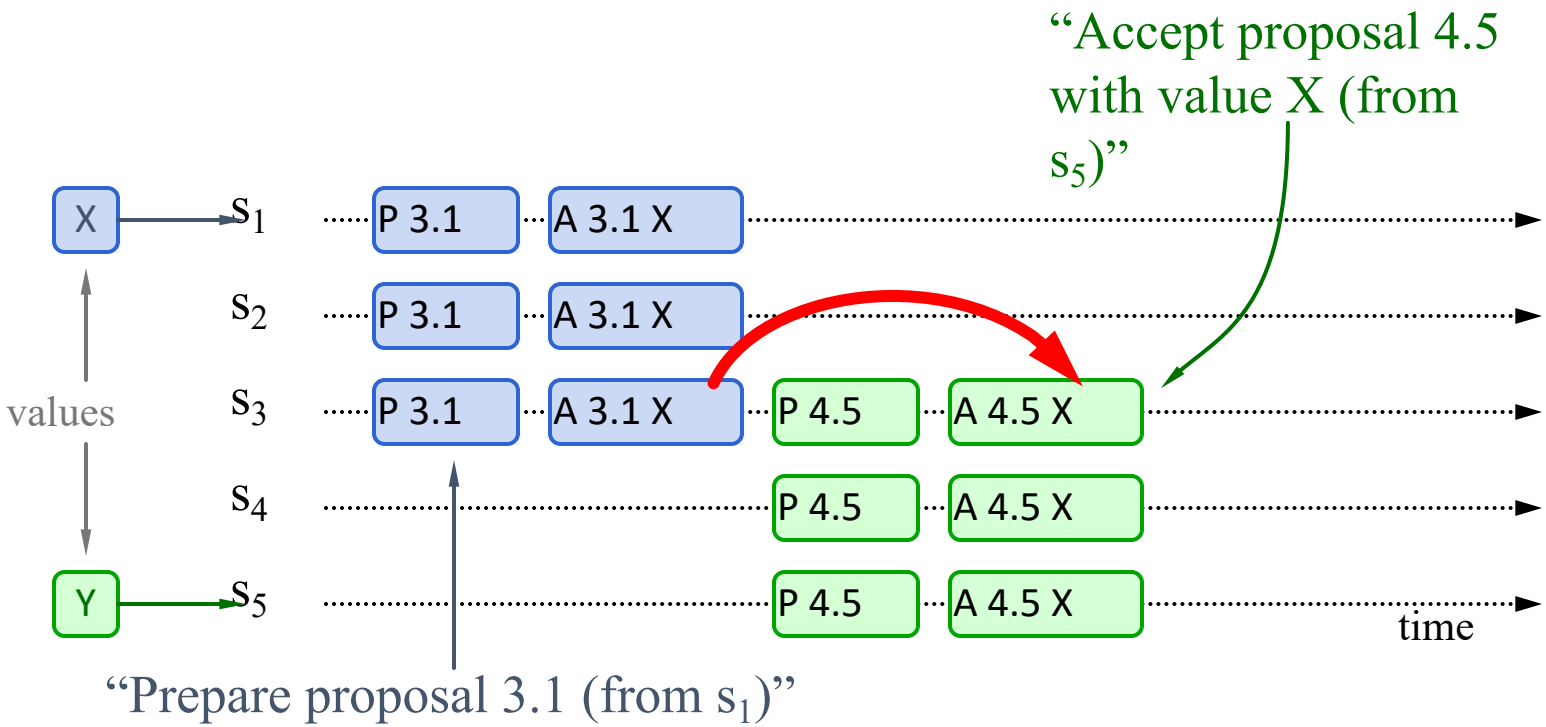
Basic Paxos


Proposers
Acceptors
1) Choose new proposal number n
2) Broadcast Prepare(n) to all servers
3) Respond to Prepare(n):
✓ If n > minProposal then minProposal = n
✓ Return(acceptedProposal, acceptedValue)
4) When responses received from majority:
✓ If any acceptedValues returned, replace value with acceptedValue for highest acceptedProposal
5) Broadcast Accept(n, value) to all servers
6) Respond to Accept(n, value):
✓ If n ≥ minProposal then acceptedProposal = minProposal = n acceptedValue = value
✓ Return(minProposal)
6) When responses received from majority:
✓ Any rejections (result > n)? goto (1)
✓ Otherwise, value is chosen

Basic Paxos Examples

Three possibilities when later proposal prepares:

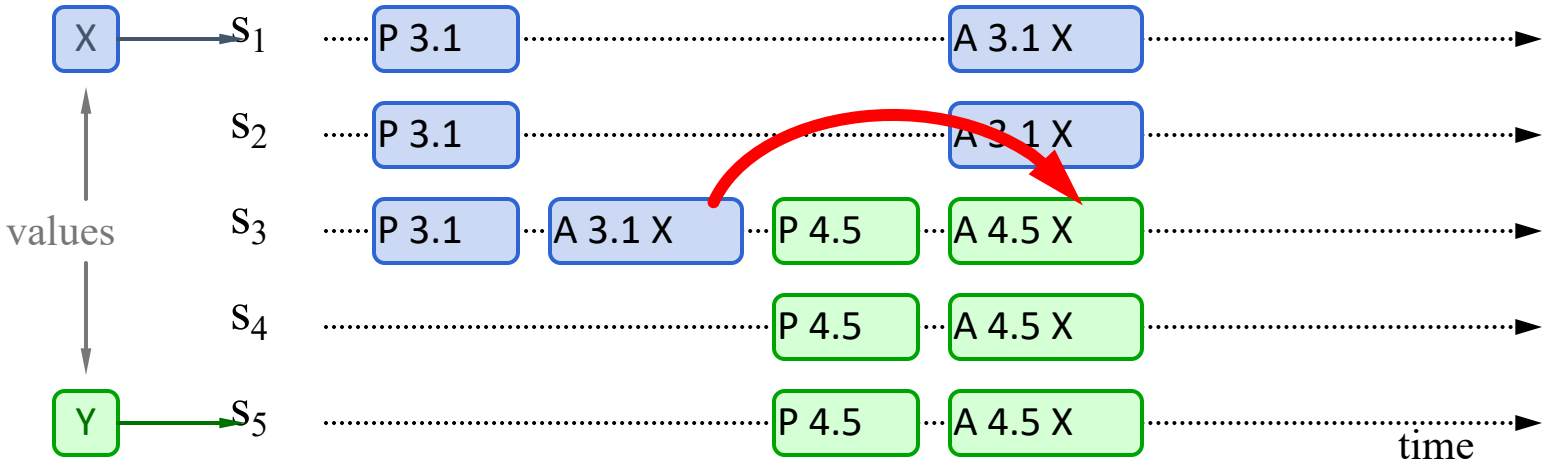
1. Previous value already chosen:
   - ✓ New proposer will find it and use it



Basic Paxos Examples, cont'd

Three possibilities when later proposal prepares:

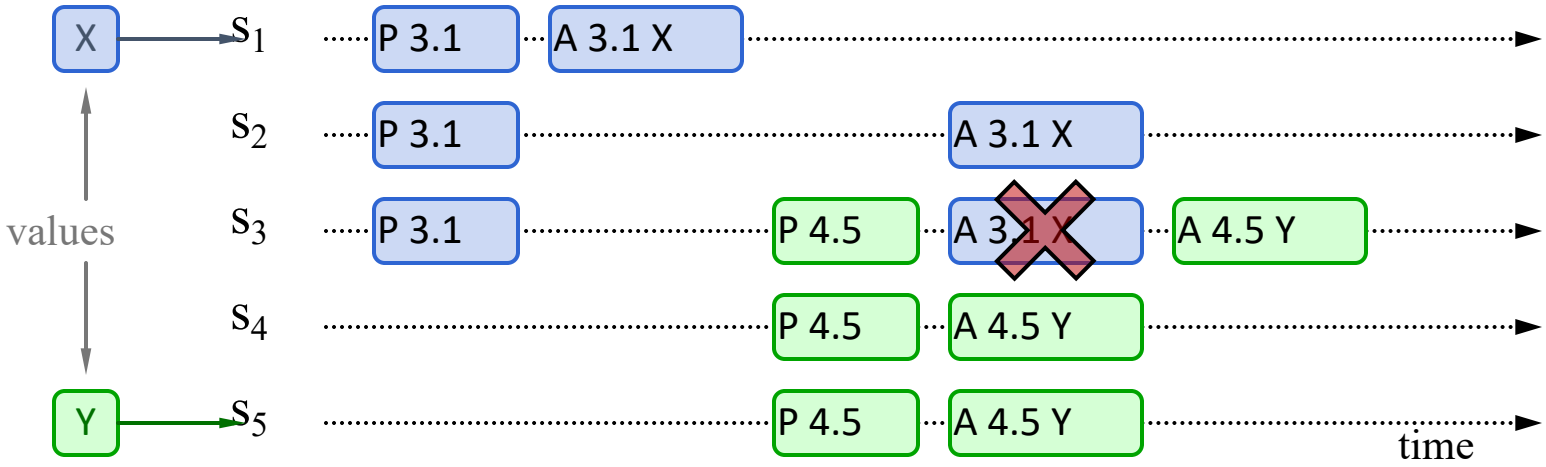
2. Previous value not chosen, but new proposer sees it:
   - ✓ New proposer will use existing value
   - ✓ Both proposers can succeed



Basic Paxos Examples, cont'd

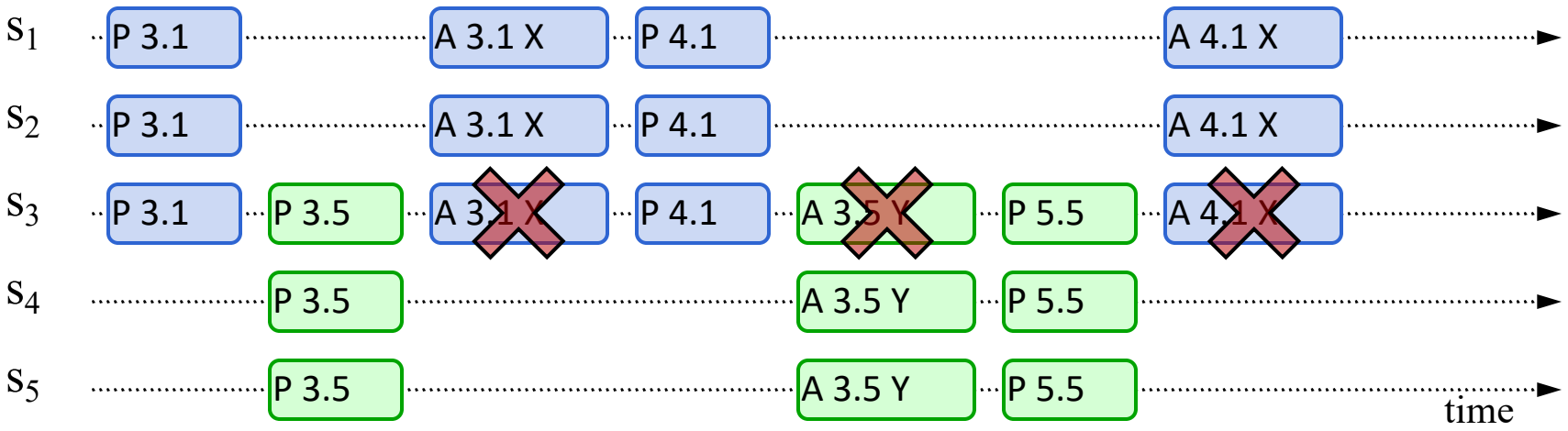
Three possibilities when later proposal prepares:

3. Previous value not chosen, new proposer doesn't see it:
   - ✓ New proposer chooses its own value
   - ✓ Older proposal blocked



Liveness

- Competing proposers can livelock:



- One solution: randomized delay before restarting
  - ✓ Give other proposers a chance to finish choosing
- Multi-Paxos will use leader election instead

Slides adapted from Ousterhout and Ongaro@Stanford