Fault Tolerance and Reliability in DS

Fault Tolerance

- Basic concepts in fault tolerance
- Masking failure by redundancy
- Process resilience

Motivation

- Single machine systems
  - Failures are all or nothing
    - OS crash, disk failures
- Distributed systems: multiple independent nodes
  - Partial failures are also possible (some nodes fail)
- *Question:* Can we automatically recover from partial failures?
  - Important issue since probability of failure grows with number of independent components (nodes) in the systems
  - Prob(failure) = Prob(Any one component fails)=1-P(no failure)

A Perspective

- Computing systems are not very reliable
  - OS crashes frequently (Windows), buggy software, unreliable hardware, software/hardware incompatibilities
  - Until recently: computer users were "tech savvy"
    - Could depend on users to reboot, troubleshoot problems
  - Growing popularity of Internet/World Wide Web
    - "Novice" users
    - Need to build more reliable/dependable systems
  - Example: what is your TV (or car) broke down every day?
    - Users don't want to "restart" TV or fix it (by opening it up)
- Need to make computing systems more reliable

Basic Concepts

- Fault – physical defect, imperfection, or flaw that occurs within hardware or software unit.
- Error – manifestation of a fault. Deviation from accuracy or correctness.
- Failure – if error results in the system performing one of its functions incorrectly.

Basic Concepts (cont'd)

- Need to build *dependable* systems
- Requirements for dependable systems
  - Availability: system should be available for use at any given time
    - 99.999 % availability (five 9s) => very small down times
  - Reliability: system should run continuously without failure
  - Safety: temporary failures should not result in a catastrophic
    - Example: computing systems controlling an airplane, nuclear reactor
  - Maintainability: a failed system should be easy to repair
  - Security: avoidance or tolerance of deliberate attacks to the system

Basic Concepts (cont'd)

- Fault tolerance: system should provide services despite 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*	The 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.

Fault types

- Node (hardware) faults
- Program (software) faults
- Communication faults
- Timing faults

- Implies types of redundancy

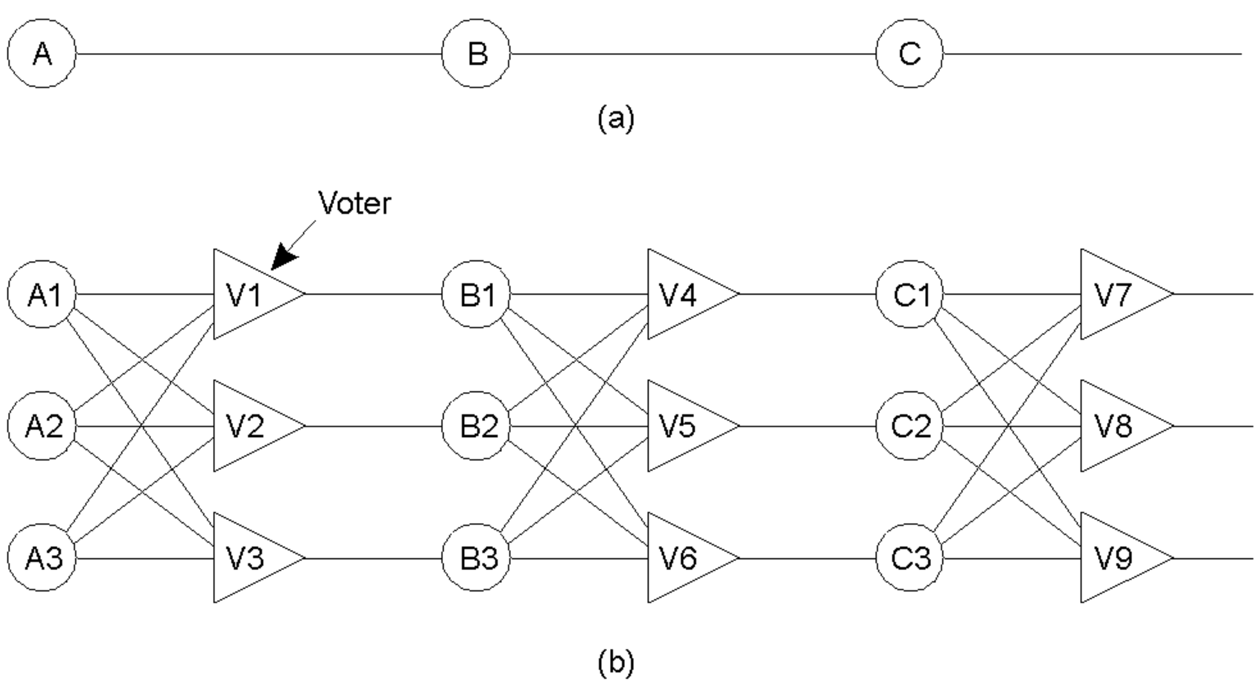
Types of redundancy

- Hardware redundancy, extra PE, I/O
- Software redundancy, extra versions of modules
- Information redundancy, error detection bits
- Time redundancy, additional time to perform functions of a system

Fault handling methods

- Active replication – all replication modules and their internal states are closely synchronized.
- Passive replication – only one module is active but other module's internal states are regularly updated by means of checkpoint from active module.
- Semi-active – hybrid of both active and passive replication. Low recovery overhead.

Failure Masking by Redundancy

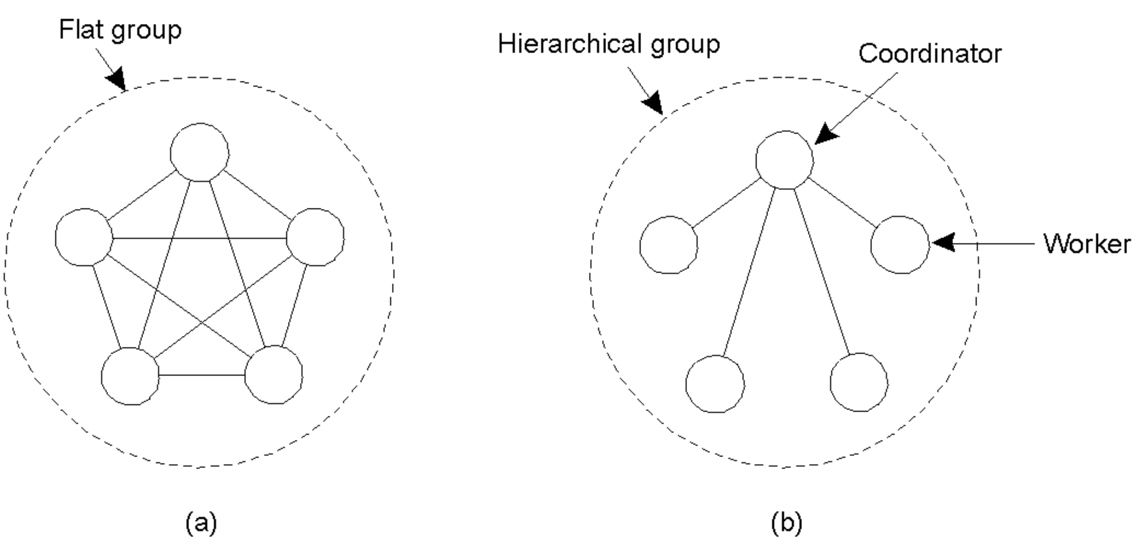


- Triple modular redundancy.

Process Resilience

- Handling faulty processes: organize several processes into a group
  - All processes perform same computation
  - All messages are sent to all members of the group
  - Majority need to agree on results of a computation
  - Ideally want multiple, independent implementations of the application (to prevent identical bugs)
- Use *process groups* to organize such processes

Flat Groups versus Hierarchical Groups



Advantages and disadvantages?

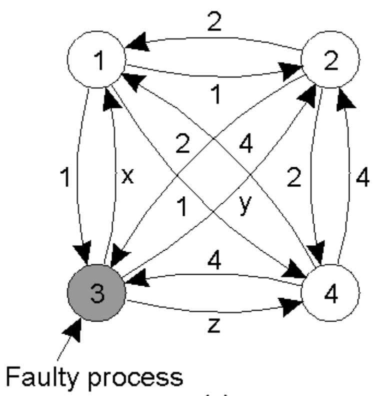
Agreement in Faulty Systems

- How should processes agree on results of a computation?
- *K-fault tolerant*: system can survive k faults and yet function
- Assume processes fail silently
  - Need (k+1) redundancy to tolerant k faults
- *Byzantine failures*: processes run even if sick
  - Produce erroneous, random or malicious replies
    - Byzantine failures are most difficult to deal with
  - Need ? Redundancy to handle Byzantine faults

Byzantine Faults

- Simplified scenario: two perfect processes with unreliable channel
  - Need to reach agreement on a 1 bit message
- Two army problem: Two armies waiting to attack
  - Each army coordinates with a messenger
  - Messenger can be captured by the hostile army
  - Can generals reach agreement?
  - Property: Two perfect process can never reach agreement in presence of unreliable channel
- Byzantine generals problem: Can N generals reach agreement with a perfect channel?
  - M generals out of N may be traitors

Byzantine Generals Problem



(a)

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)

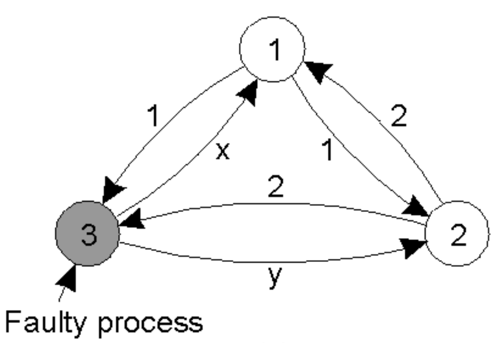
1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

- Recursive algorithm by Lamport
- The Byzantine generals problem for 3 loyal generals and 1 traitor.

a) The generals announce their troop strengths (in units of 1 kilosoldiers).
b) The vectors that each general assembles based on (a)
c) The vectors that each general receives in step 3.

Byzantine Generals Problem Example



(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)

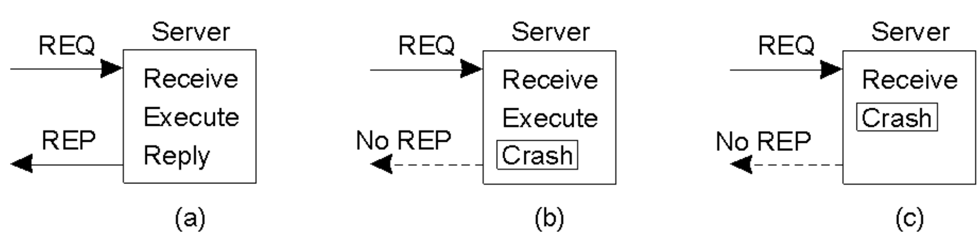
- The same as in previous slide, except now with 2 loyal generals and one traitor.
- Property: With *m* faulty processes, agreement is possible only if 2*m*+1 processes function correctly [Lamport 82]
  - Need more than two-thirds processes to function correctly

More on Fault Tolerance

- Reliable communication
  - One-one communication
  - One-many communication
- Distributed commit
  - Two phase commit
  - Three phase commit
- Failure recovery
  - Checkpointing
  - Message logging

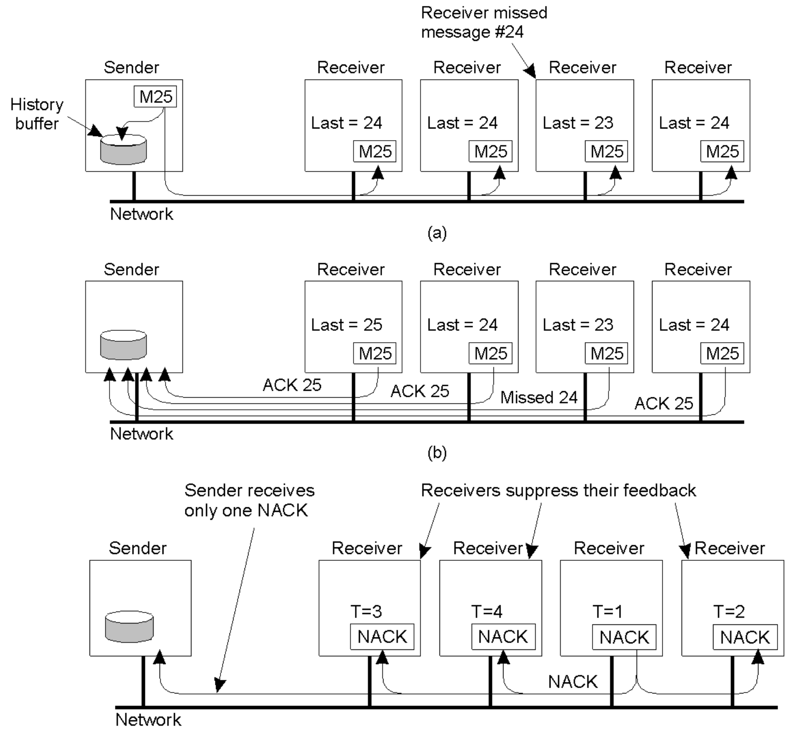
Reliable One-One Communication

- Issues were discussed in Lecture 3
  - Use reliable transport protocols (TCP) or handle at the application layer
- RPC semantics in the presence of failures
- Possibilities
  - Client unable to locate server
  - Lost request messages
  - Server crashes after receiving request
  - Lost reply messages
  - Client crashes after sending request



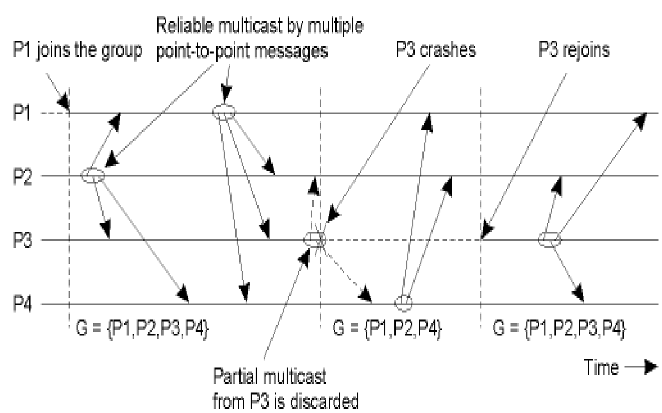
Reliable One-Many Communication

- Reliable multicast
  - Lost messages => need to retransmit
- Possibilities
  - ACK-based schemes
    - Sender can become bottleneck
  - NACK-based schemes



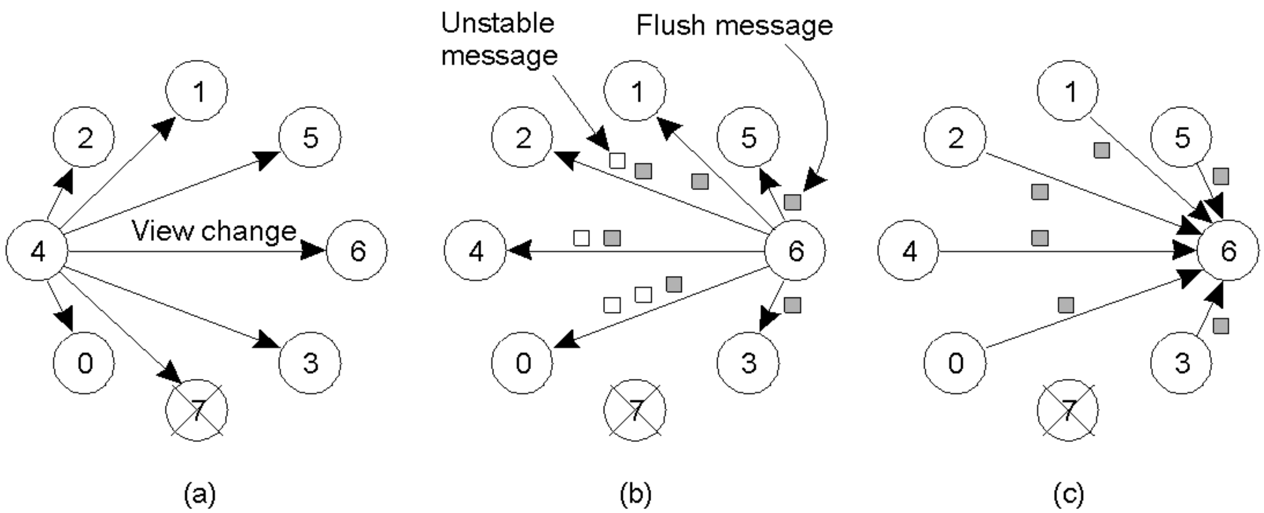
Atomic Multicast

- Atomic multicast: a guarantee that all process received the message or none at all
  - Replicated database example
- Problem: how to handle process crashes?
- Solution: *group view*
  - Each message is uniquely associated with a group of processes
    - View of the process group when message was sent
    - All processes in the group should have the same view (and agree on it)



Virtually Synchronous Multicast

Implementing Virtual Synchrony in Isis



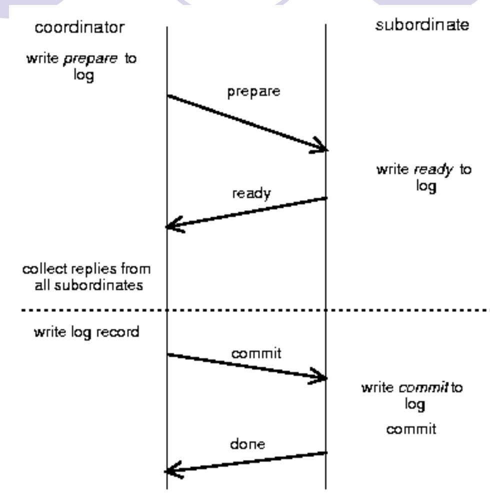
a) Process 4 notices that process 7 has crashed, sends a view change
b) Process 6 sends out all its unstable messages, followed by a flush message
c) Process 6 installs the new view when it has received a flush message from everyone else

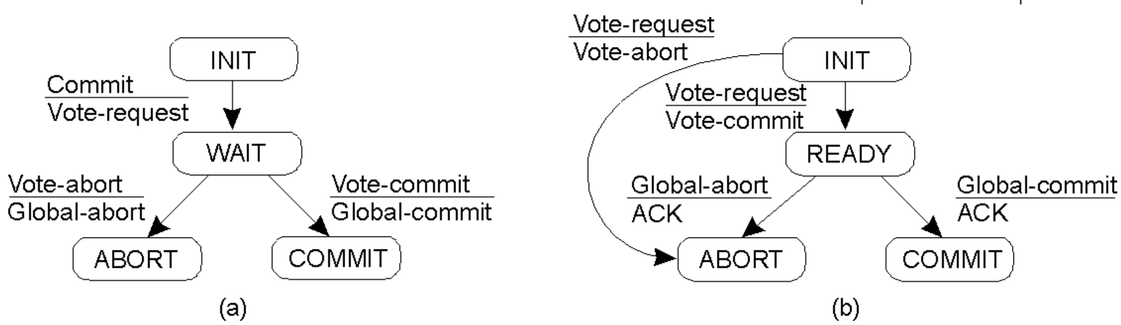
Distributed Commit

- Atomic multicast example of a more general problem
  - All processes in a group perform an operation or not at all
  - Examples:
    - Reliable multicast: Operation = delivery of a message
    - Distributed transaction: Operation = commit transaction
- Problem of distributed commit
  - All or nothing operations in a group of processes
- Possible approaches
  - Two phase commit (2PC) [Gray 1978]
  - Three phase commit

Two Phase Commit

- Coordinator process coordinates the operation
- Involves two phases
  - Voting phase: processes vote on whether to commit
  - Decision phase: actually commit or abort





Implementing Two-Phase Commit

actions by coordinator:

```
while 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 {
        while 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;
}
```

- Outline of the steps taken by the coordinator in a two phase commit protocol

Implementing 2PC

actions by participant:

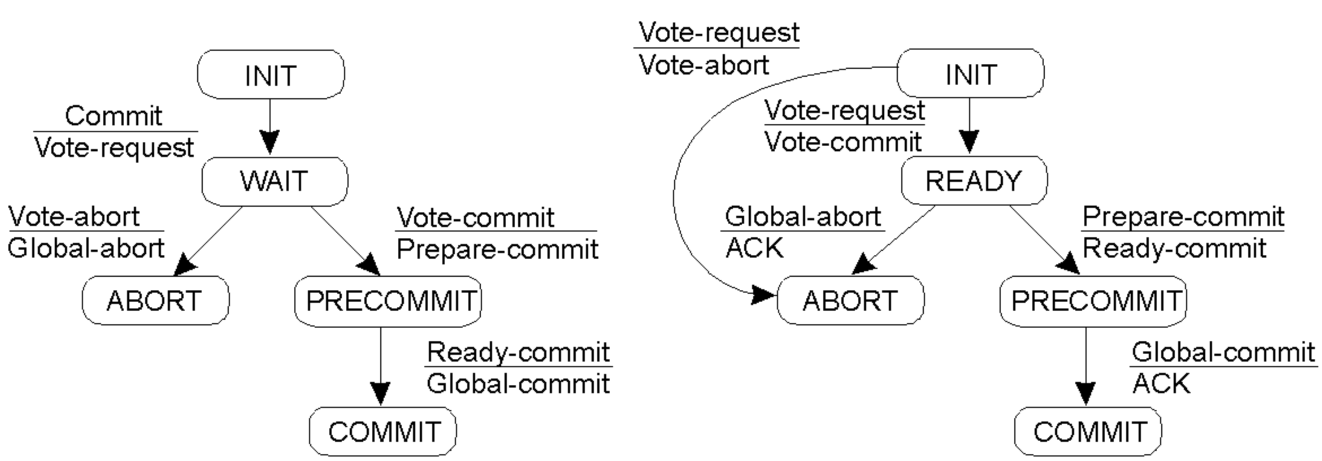
```
write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
   write VOTE_ABORT to local log;
   exit;
}
if participant votes COMMIT {
   write VOTE_COMMIT to local log;
   send VOTE_COMMIT to coordinator;
   wait for DECISION from coordinator;
   if timeout {
      multicast DECISION_REQUEST 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;
}
```

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 */
```

Three-Phase Commit



(a) (b)

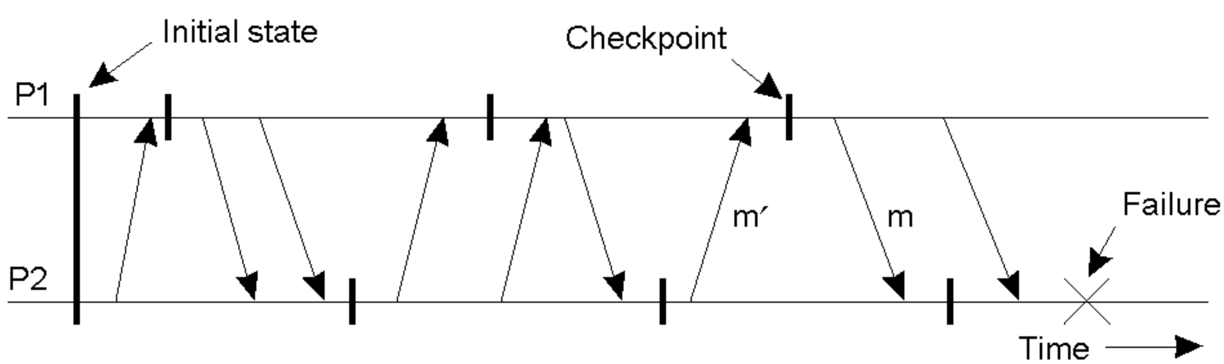
Two phase commit: problem if coordinator crashes (processes block)

Three phase commit: variant of 2PC that avoids blocking

Recovery

- Techniques thus far allow failure handling
- Recovery: operations that must be performed after a failure to recover to a correct state
- Techniques:
  - Checkpointing:
    - Periodically checkpoint state
    - Upon a crash roll back to a previous checkpoint with a *consistent state*

Independent Checkpointing



- Each processes periodically checkpoints independently of other processes
- Upon a failure, work backwards to locate a consistent cut
- Problem: if most recent checkpoints form inconsistent cut, will need to keep rolling back until a consistent cut is found
- Cascading rollbacks can lead to a domino effect.

Coordinated Checkpointing

- Take a distributed snapshot
- Upon a failure, roll back to the latest snapshot
  - All process restart from the latest snapshot

Message Logging

- Checkpointing is expensive
  - All processes restart from previous consistent cut
  - Taking a snapshot is expensive
  - Infrequent snapshots => all computations after previous snapshot will need to be redone [wasteful]
- Combine checkpointing (expensive) with message logging (cheap)
  - Take infrequent checkpoints
  - Log all messages between checkpoints to local stable storage
  - To recover: simply replay messages from previous checkpoint
    - Avoids recomputations from previous checkpoint