CprE 450/550x
Distributed Systems and Middleware

Fault Tolerance

Yong Guan
3216 Coover
Tel: (515) 294-8378
Email: guan@ee.iastate.edu
April 22 & 27, 2004

Readings for Today's Lecture

- References
  - Chapter 7 of "Distributed Systems: Principles and Paradigms"

3

Basic Concepts

- Availability
- Reliability
- Safety
- Maintainability

- Security

4

Basic Concepts (cont.)

- A system is said to fail when it cannot meet its promises
- An error is a part of a system's state that may lead to a failure
- The cause of an error is called fault

- Building dependable systems closely relates to controlling faults
- Fault tolerance means that a system can provide its services even in the presence of faults

5

Basic Concepts (cont.)

- Transient fault
- Intermittent fault
- Permanent fault

6

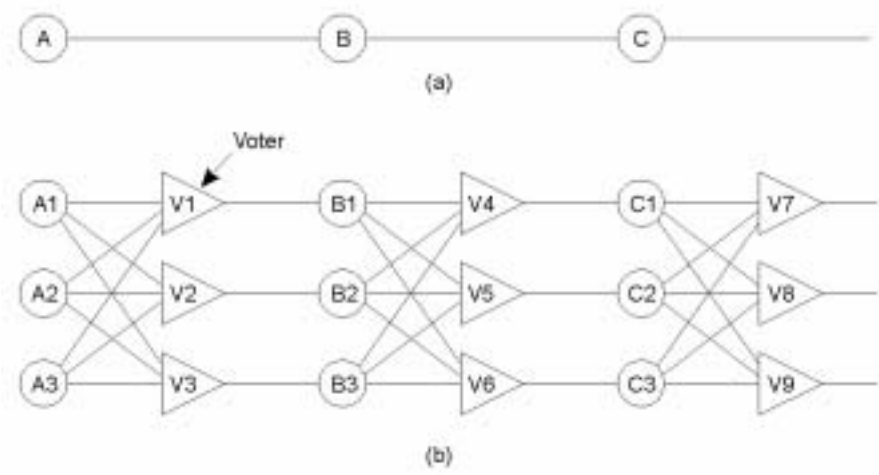
Failure Models

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.

7

Failure Masking by Redundancy



Triple modular redundancy. (TMR)

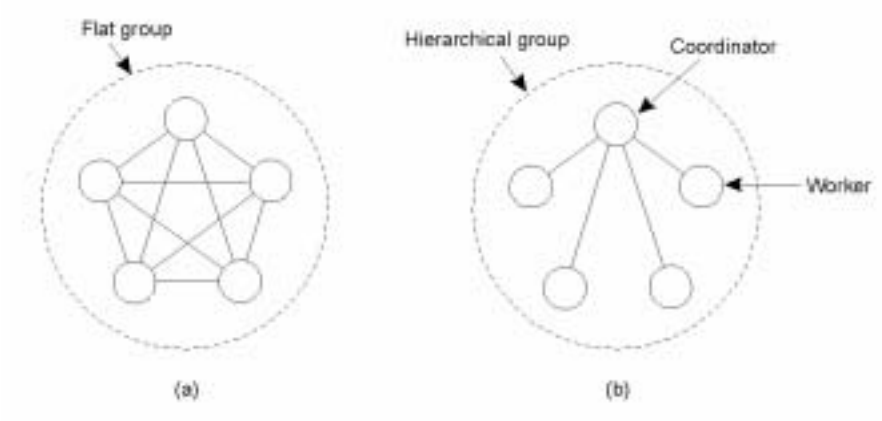
8

Process Resilience

- Design issues
- Group Memberships
- Failure Masking and Replication
  - **Primary-based replication**
  - **Replication-write protocols (quorum-based protocols, etc.)**
- K fault tolerance
- Agreement in Faulty Systems

9

Flat Groups versus Hierarchical Groups

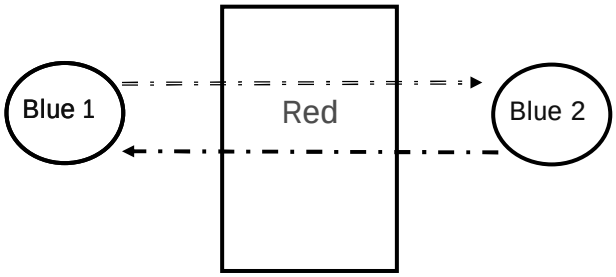


a) Communication in a flat group.
b) Communication in a simple hierarchical group

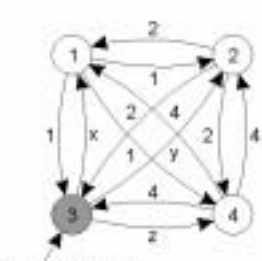
10

Agreement in Faulty Systems

- The general goal: Have all the non-faulty processes reach consensus on some issues and establish it within a finite number of steps.
- Example: Two-army problem



Agreement in Faulty Systems (1)



(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)

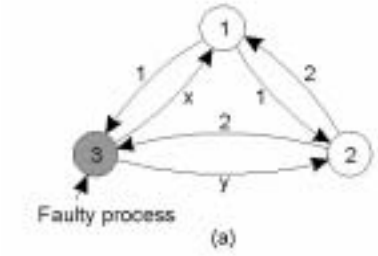
1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

The Byzantine generals problem for 3 loyal generals and1 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.

Agreement in Faulty Systems (2)



(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.

13

Agreement in Faulty Systems

- Lamport (1983)

Given m faulty processes, agreement can be achieved only if 2m+1 correctly functioning processes are present, for a total of 3m+1 processes.

14

15

Reliable Client-Server Communication

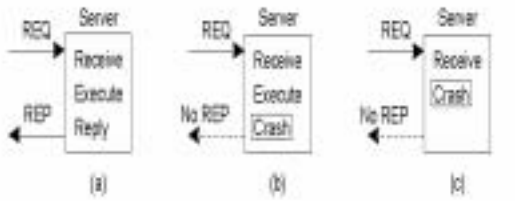
- A communication channel may exhibit crash, omission, timing, and arbitrary failures.
- Reliable transport protocol: TCP
- RPC semantics in the presence of failures
  - **RPC: hide communication by making remote procedure calls as local ones**
  - **If both C and S work perfectly, RPC does its job well.**
  - **Problem:** It is not easy to mask the difference between remote and local calls in the presence of failures**.**

16

Reliable Client-Server Communication

- Five classes of RPC failures:
  1. **The client is unable to locate the server**
  2. **The request message from the client to the server is lost**
  3. **The server crashes after receiving a request**
  4. **The reply message from the server to the client is lost**
  5. **The client crashes after sending a request**

RPC: Server Crashes (1)



A server in client-server communication

a) Normal case
b) Crash after execution
c) Crash before execution

Guarantee nothing

At least once semantics

At most once semantics

Exactly once semantics

RPC: Server Crashes (2)

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

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

19

RPC: Lost reply message

- Idempotent: Some operations can safely be repeated as often as necessary with no damage being done.

20

RPC: Client Crashes

- What happens if a client sends a request to a server to do some work and crashes before the server replies?
- Orphan: At a point, a computation is active and no parent is waiting for the result. Such unwanted computation is called orphan.
  - **Waste CPU cylces**
  - **Can lock files or otherwise tie up valuable resources**

21

RPC: Client Crashes

- Four things can be done:
  - Extermination: Log what is about to do before RPC stub sends a RPC message. After reboot, the log is checked and the orphan is explicitly killed off.
  - Reincarnation: Divide time up into sequentially numbered epochs. After reboot, it broadcasts a message to all machines declaring the start of a new epoch. Once such broadcast mesg received, all remote computations on behalf of that client are killed.
  - Gentle reincarnation: When epoch broadcast comes in, each machine checks to see if it has any remote computations, and if so, tries to locate their owner. Only if the owner cannot be found is the computation killed.
  - Expiration: Each RPC is given a time T to do its job.

22

Reliable Group Communication

- TCP offers reliable point-to-point channels.
- How to implement reliable group communication?

- One way: Let each process set up a point-to-point connection to each other process it wants to.

  Not efficient

- What do we mean "Reliable Group Communication"?

  A message that is sent to a process group should be delivered to each member of that group.

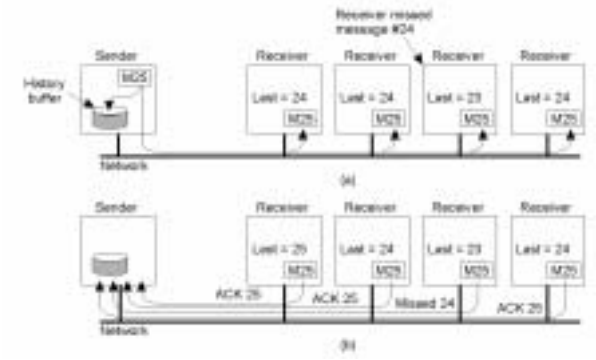
23

Reliable Group Communication

- Need further definition for “Reliable Group Communication”
  - What about new member joining the group during the communication?
  - What about an existing member leaving the group? Crashes?
- In the presence of faulty processes, multicasting is considered to be reliable when it can be guaranteed that all non-faulty group members receive the message. Agreement should be reached.

24

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

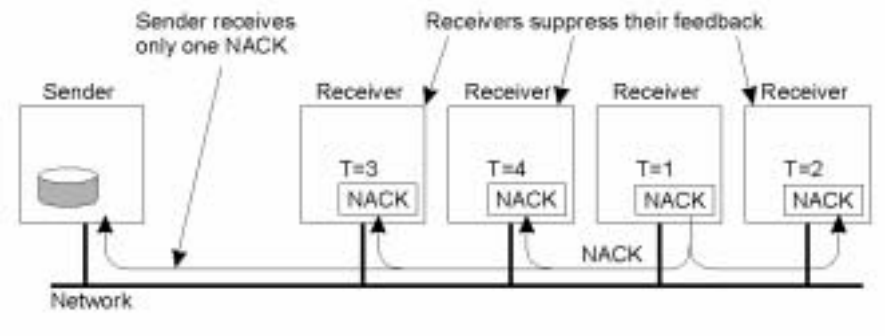
25

Scalability in Reliable Multicasting

- Problem: Feedback implosion
- Will negative acknowledgement solve the problem?
- Is there any problem with only returning negative acknowledgement?
- Feedback suppression

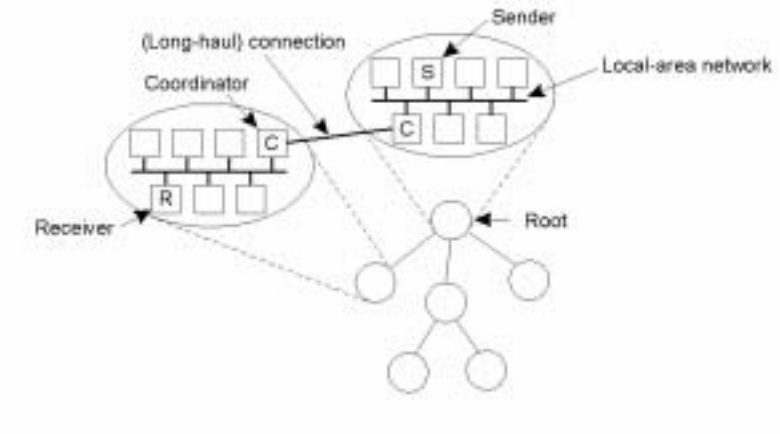
26

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.

a) Each local coordinator forwards the message to its children.
b) A local coordinator handles retransmission requests.

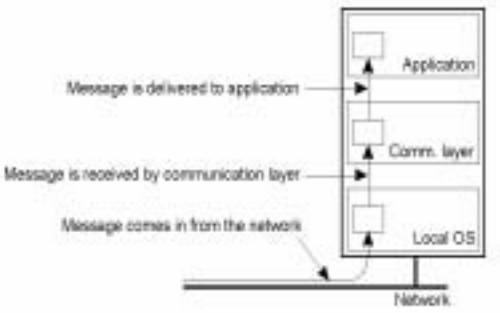
Atomic Multicast

- Atomic multicast problem: A message is delivered to either all processes or to none at all. In addition, all messages are delivered in the same order to all processes
- Why this important? Example: Replicated database

29

Virtual Synchrony (1)

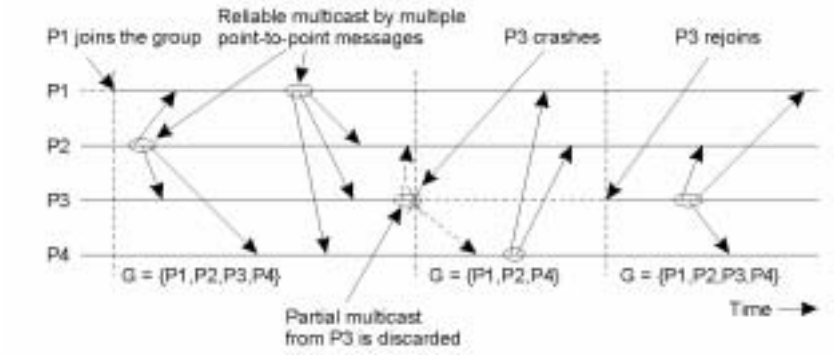
- Group View: Delivery list, the set of processes contained in the group
- View Change
- Virtually Synchronous: A mesg multicast to group view G is delivered to each non-faulty process in G. If the sender of the mesg crashes during the multicast, the mesg may either be delivered to all remaining processes or ignored by each of them.



The logical organization of a distributed system to distinguish between message receipt and message delivery

30

Virtual Synchrony (2)



The principle of virtual synchronous multicast.

31

Message Ordering (1)

Four different ordering:

- Unordered multicasts
- FIFO-ordered multicasts
- Causally-ordered multicasts
- Totally-ordered multicasts

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.

32

Message Ordering (2)

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

33

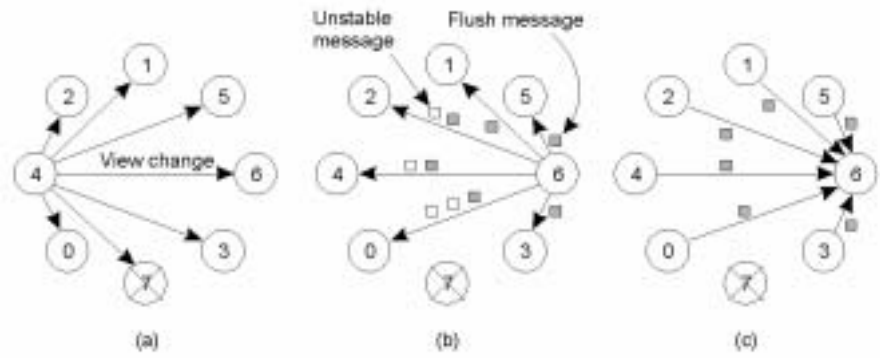
Implementing Virtual Synchrony (1)

Multicast	Basic Message Ordering	Total-ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

- Six different versions of virtually synchronous reliable multicasting.

34

Implementing Virtual Synchrony (2)



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

35

Distributed Commit

- The distributed commit problem involves having an operation being performed by each member of a process group, or none at all.
- One-phase commit protocol

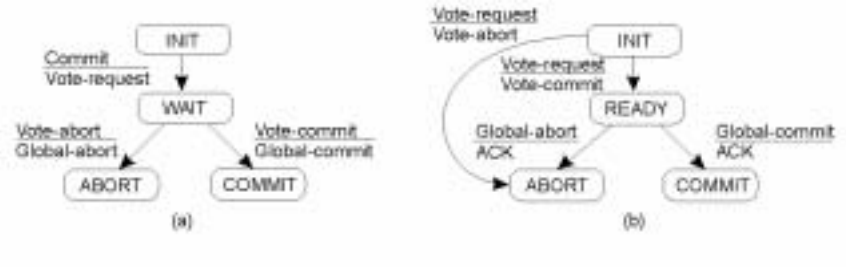
  Problem: if one of the participant can not perform the operation, no way to tell the coordinator
- Two phase commit protocol
- Three phase commit protocol

36

Two Phase Commit Protocol

1. The coordinator sends a VOTE_REQUEST message to all participants
2. When a participant receives VOTE_REQUEST, it returns either VOTE_COMMIT or VOTE_ABORT to the coordinator.
3. The coordinator collects all votes from the participants. If all participants have voted to commit the transaction, then so will the coordinator. In that case, it sends a GLOBAL_COMMIT to all participants. Otherwise, it multicasts a GLOBAL_ABORT.
4. Each participant that voted for a commit waits for the final reaction by the coordinator. Locally commit if a GLOBAL_COMMIT is received or abort if GLOBAL_ABORT is received.

Two-Phase Commit (1)



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

Two-Phase Commit (2)

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*.

39

Two-Phase Commit (3)

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

40

Two-Phase Commit (4)

- Steps taken by participant process in 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;
}
```

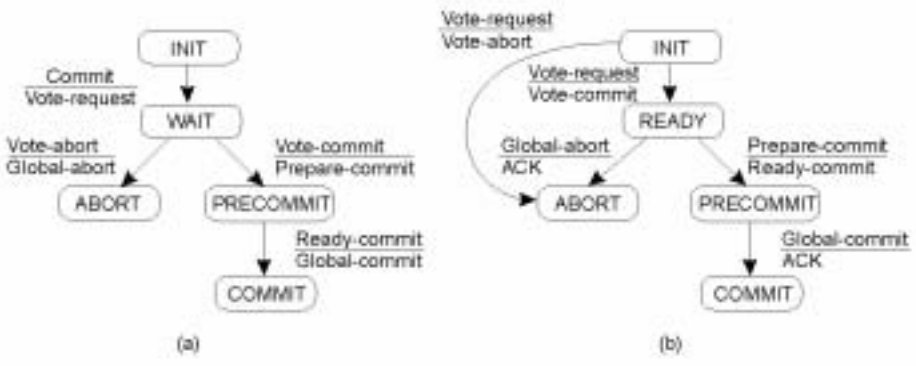
Two-Phase Commit (5)

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 */
```

Steps taken for handling incoming decision requests.

Three-Phase Commit



a) Finite state machine for the coordinator in 3PC
b) Finite state machine for a participant

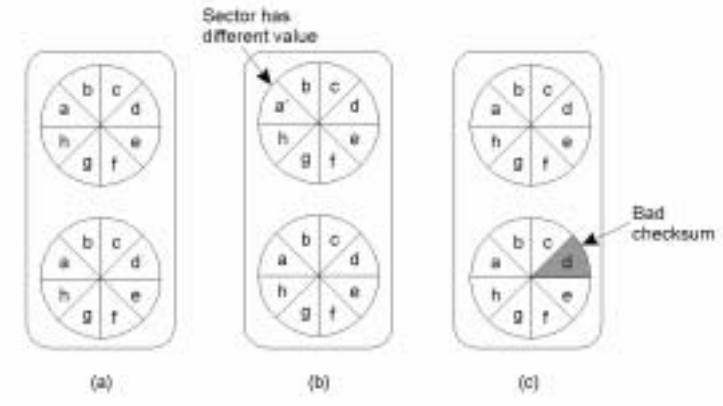
43

Recovery

- Backward recovery
  - **Checkpoint**
- Forward recovery

44

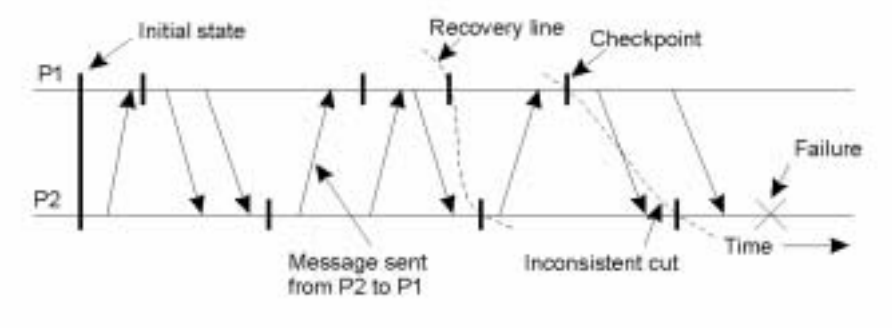
Recovery Stable Storage



a) Stable Storage
b) Crash after drive 1 is updated
c) Bad spot

45

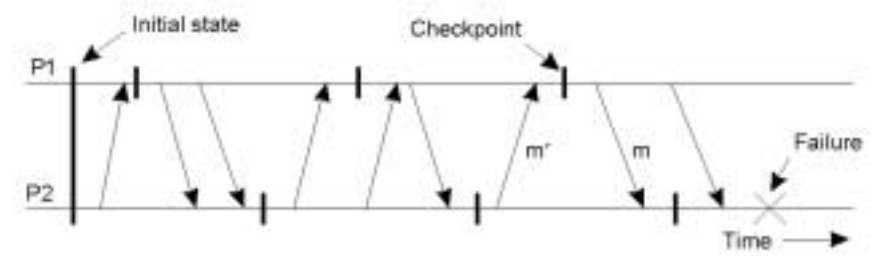
Checkpointing



A recovery line.

46

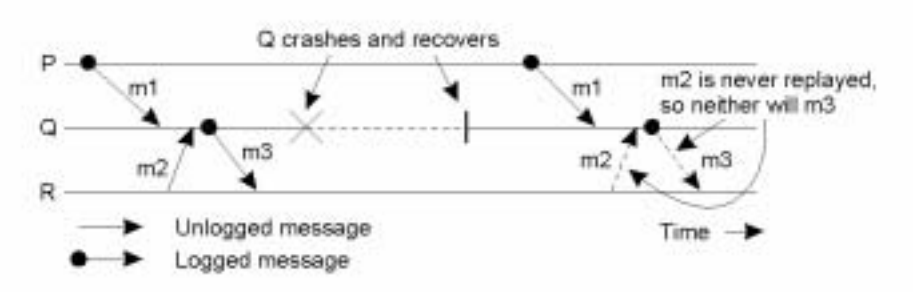
Independent Checkpointing



The domino effect.

47

Message Logging



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

48

Message Logging Scheme

- A message is stable if it can no longer be lost.
- Pessimistic logging protocols
- Optimistic logging protocols

Any Questions?

See you next time.