COP 6611 Advanced Operating System

Fault Tolerance

Chi Zhang

czhang@cs.fiu.edu

Basic Concepts

Dependability Includes

- Availability
  - Run time / total time
- Reliability
  - The length of uninterrupted run time
- Safety
  - When a system temporarily fails, nothing catastrophic happens
- Maintainability
  - repairable

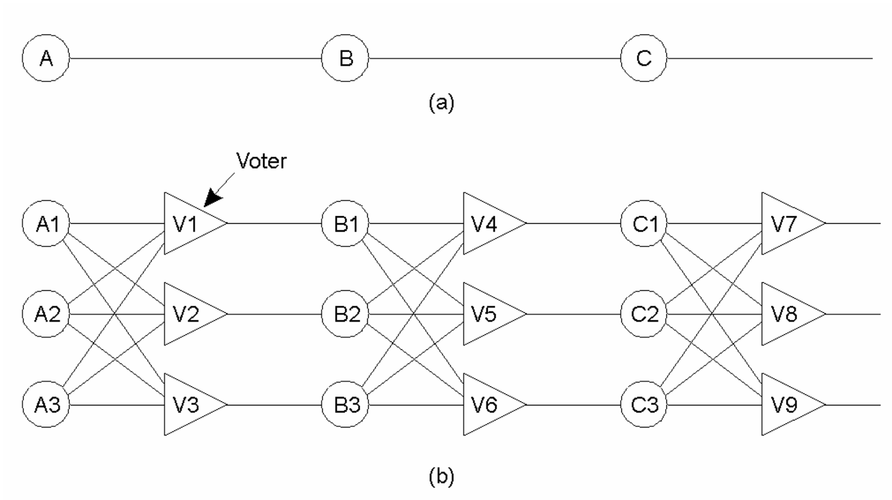
Failure Models

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts (reboot!)
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 (for real-time performance)	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 (even malicious, intentional)

Fault-tolerance: a system can provide its services even in the presence of faults

Fail-stop (detectable), Fail-Silent, and Fail-Safe (recognizable junk)

Failure Masking by Redundancy



(a)

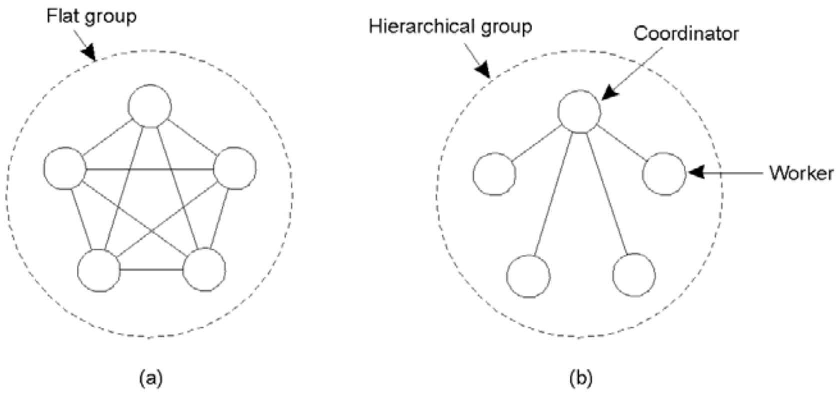
(b)

Triple modular redundancy.

Process Resilience

- What if a process fails? ⇒ A group of identical process
- When a message is sent to a group, all members receive it.
- Group management: join / leave.
  - Centralized / Distributed
- Discover the crashed processes
- Data Replication Management
  - Primary-based
  - Replicated-based

Flat Groups versus Hierarchical Groups



a) Communication in a flat group. No single point of failure.

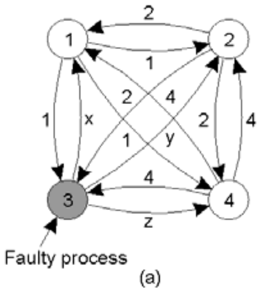
b) Communication in a simple hierarchical group. Decision making is less complicated

Agreement in Faulty Systems (1)

Goal: non-faulty processes reach consensus in finite steps.

Unreliable Communication ⇒ No agreement between 2 processes

What if processes are faulty?



(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.

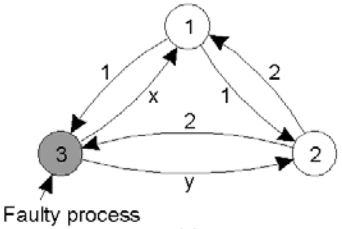
a) The generals announce their troop strengths (in units of 1 kilosoldiers).

b) The vectors that each general assembles based on (a)

c) Every general passes his vector to every other general.

Finally, take the majority or mark UNKOWN.

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)

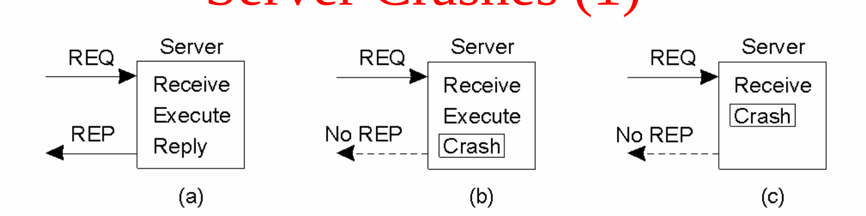
(c)

The same as in previous slide, except now with 2 loyal generals and one traitor.

Reliable Client-Server Communication

- TCP masks omission failures by ACKs and re-trans.
- Distributed systems automatically setup a new connection after a crash failure.
- RPC might face five classes failures:
  - Unable to locate the server
  - The request message is lost. E.g. Server crashes before receiving the request. Timeout ⇒ Retransmit.
  - The reply message is lost. ⇒ Message IDs. (whether request or reply is lost?)
  - The server crashes after receiving the request
    - When a server crashes, it loses all states!
  - The client crashes after sending a request.
    - Kill the orphan process that wastes resources.
    - In a single computer, clients and servers crash simultaneously.

Server Crashes (1)



A server in client-server communication

a) Normal case

b) Crash during / after execution ⇒ no more execution!

c) Crash before execution ⇒ the client retransmit

How to distinguish (b) and (c)?

4 client-side strategies and 2 server-side strategies.

Semantics:

(i) At least once. (ii) At most once. (iii) No Guarantee.

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

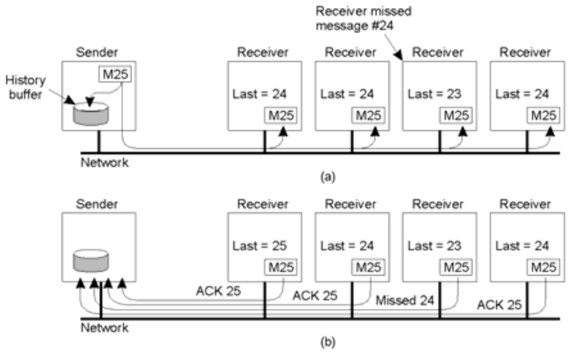
client and server strategies in the presence of server crashes.

M: Send Reply Message; P: Print; C: Crash

OK: Print Once; DUP: Print more than once

ZERO: Not printed.

Basic Reliable-Multicasting Schemes



A simple solution to reliable multicasting

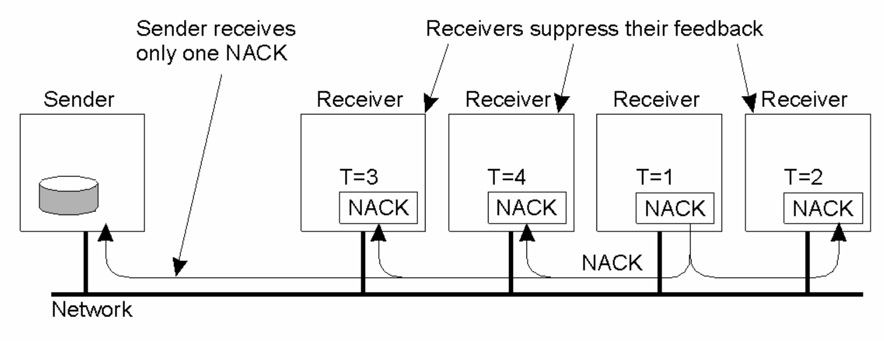
a) Message transmission. (Keep messages in buffer until each receivers ack)

b) Reporting feedback

Not scalable: feedback explosion.

One solution: Negative ACKs only. But how long to keep the message?

Nonhierarchical Feedback Control

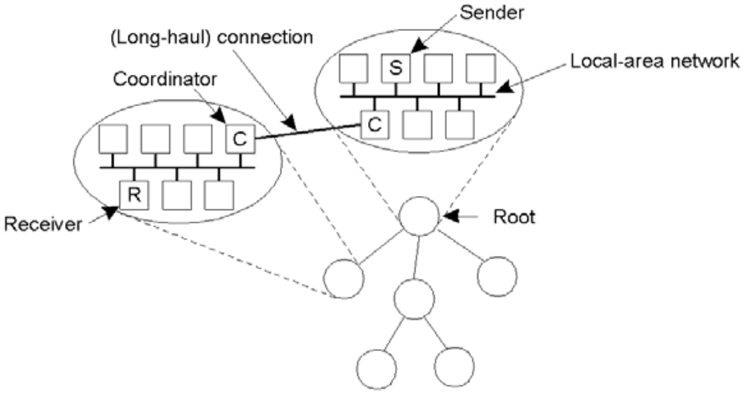


Several receivers have scheduled a request for retransmission, but the first retransmission request leads to the suppression of others.

Receivers multicast their NACK to the rest of the group after some random delay. (How to set the timer?)

Receiver might assist in local recovery.

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.

Multicast routers can server as coordinators

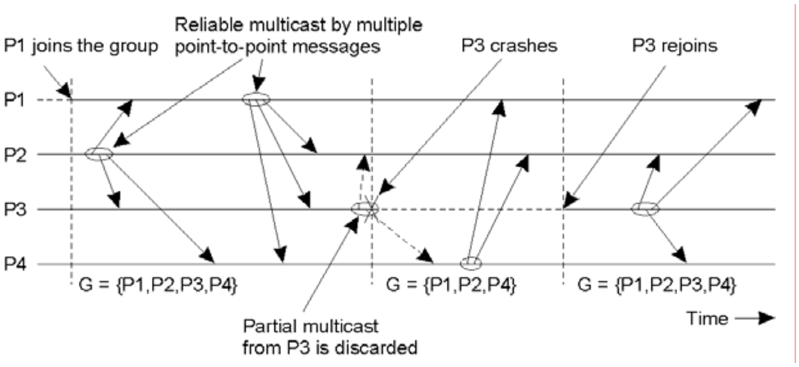
Atomic Multicast

- Messages are delivered to all processes or none.
  - Processes might crash.
  - A message is sent to all replicas just before one of them crashes is either delivered to all non-faulty processes, all none at all.
    - Non-trivial if the message is sent out by the crashed process.
  - When the crashed process recovers and rejoins the group, its state is brought up-to-date.
- Totally-ordered

Virtual Synchrony (1)

- Group view: a list of processes
  - Each message is associated with a group view.
- Suppose message *m* is sent out with group view *G*. Meanwhile a view change message *vc* is sent simultaneously. Either
  - *m* is delivered to *all* non-faulty processes in *G before* each one of them is delivered *vc*.
    - Not after, because *m* is associated with *G*.
  - *m* is not delivered at all.
    - Non-trivial if the sender of *m* crashes. (**Virtual Synchrony**).

Virtual Synchrony (2)



The principle of virtual synchronous multicast.

Message Ordering (1)

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

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

Message Ordering (2)

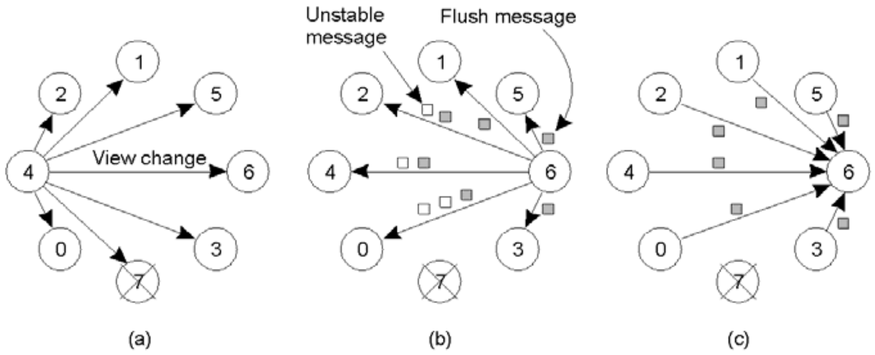
Process P1	Process P2	Process P3	Process P4
sends m1	receives m1	receives m3	sends m3
sends m2	receives m3	receives m1	sends m4
	receives m2	receives m2	
	receives m4	receives m4	

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

Implementing Virtual Synchrony (1)

- Use reliable point-to-point communication (TCP)
- Messages sent by the same process are delivered to another process in the same order.
- A message is not delivered immediately after it is received (*unstable* message).
- Protocol (p. 392)
  - When a coordinator receives a view-change initiation, it forwards a copy of all unstable messages in the current view to all processes. It then multicasts a flush message for the new group view.

Implementing Virtual Synchrony (2)

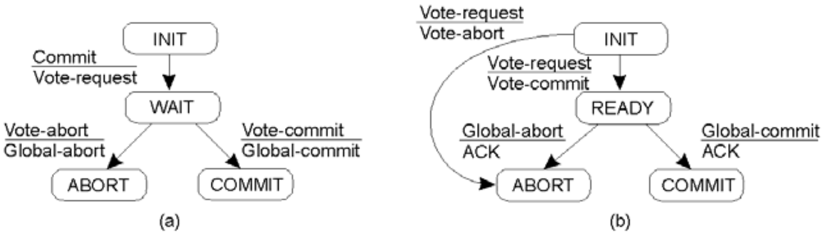


a) Process 4 notices that process 7 has crashed, sends a view change

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

Two-Phase Commit (1)



2PC: Coordinator and Participants. Phase 1: vote; Phase 2: Decision (p. 394).

- The finite state machine for the coordinator .
- The finite state machine for a participant.

Two-Phase Commit (2)

Crashed processes: States have been saved as logs.

- P recovers to *INIT* ⇒ abort.
- P recovers to *READY* ⇒ retransmit or waits (see the next slide)
- C recovers to *WAIT* ⇒ retransmit vote requests or abort
- C recovers to *COMMIT* / *ABORT* ⇒ retransmit the decision.
  - write commit / abort logs first and then multicast decision messages
  - Why force write: what if C crashes after sending decisions to *some* Ps and recovers to *WAIT*?

Two-Phase Commit (3)

Waiting processes: actions upon timeout?

- P waits in *INIT* ⇒ abort.
- C waits in *WAIT* ⇒ abort.
- P in *READY*
  - Already voted yes, can't simply abort!
  - Other participants or C might vote no.
  - Wait until C recovers. (blocking 2PC)
  - P may contact another participant Q.

Two-Phase Commit (4)

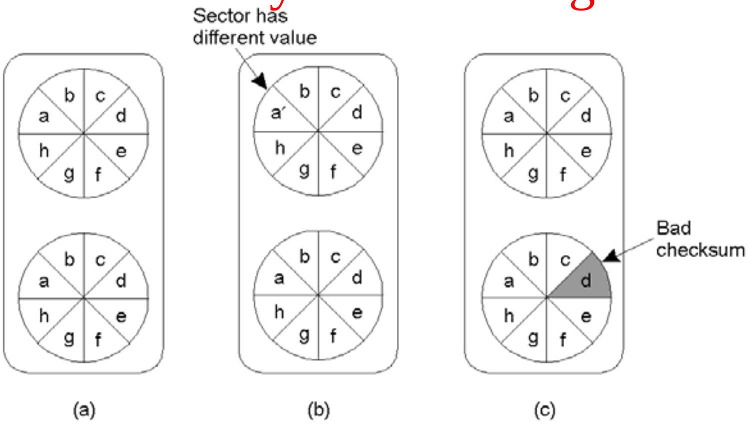
State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Wait and contact another participant

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

Recovery

- The recovery of general purpose process
  - vs. 2PC for specific scenario (distributed database)
- The system's state is periodically recorded (checkpoints)
  - A costly operation
- Message logging
  - The receiver process logs a message before it is delivered.
  - Recover to the latest check point, and then replay the messages delivered after awards.
  - Assumption: messages are the only non-deterministic factors in the system.
  - Not necessarily forced.
- Problem: Consistency among recovered processes.
  - Messages must have been sent before it is received.

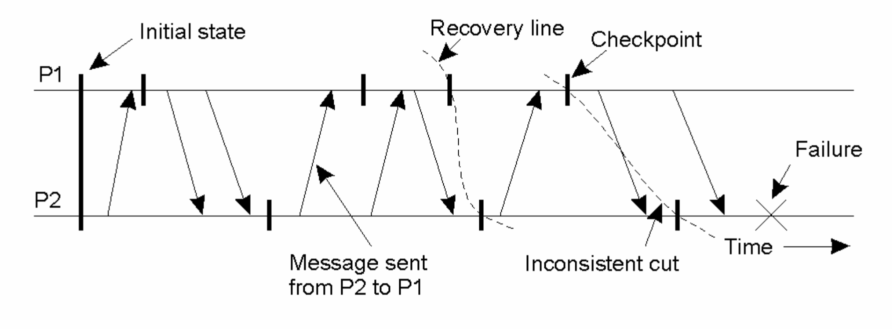
Recovery Stable Storage



Write Two (always disk 1 first) and Read One

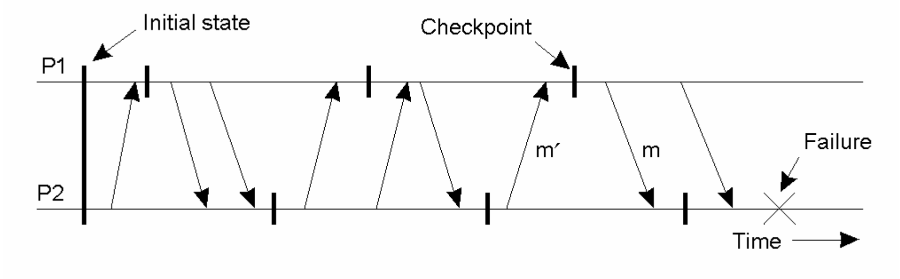
- Stable Storage
- Crash after drive 1 is updated (copy disk 1 to disk 2)
- Bad spot (copy from another disk)

Checkpointing



A consistent recovery line.

Independent Checkpointing

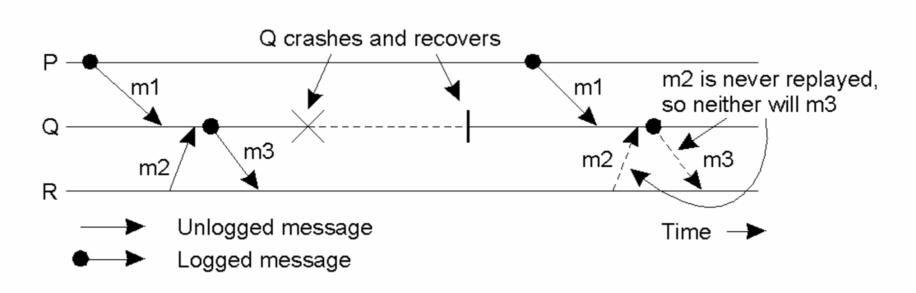


If processes take checkpoints independently, jointly roll back to a consistent line. The domino effect: In this case, the consistent recovery line is the initial state

Coordinated Checkpointing

- Independent Check Pointing
  - More checkpoints need to be maintained.
    - May recover to the initial state!
  - Complexity in computing the recovery line
    - Piggyback *(i, m)* into the messages, where *i* is the process ID, and *m* is the number of the next checkpoint on process P_i.
    - If P_i recovers to its checkpoint *m-1*, all other processes must recover to a checkpoint before any messages with tag *(i, n) (n ≥m)* is received.
- Coordinate checkpoiting
  - Automatically consistent: distributed snapshot algorithm.

Message Logging (1)



Incorrect replay of messages after recovery:

m1 is replayed by Q but m2 is not. R sees m3, which is sent because Q reads m2.

R is an orphan process

The issue: some message logs must be written to the stable storage (otherwise lost after recover)

Message Logging (2)

- DEP(m): the set of processes that are dependent on m.
- COPY(m): the set of processes that have delivered m or have a copy of m but not yet in their stable storage.
- Process Q is an orphan if Q is in DEP(m) while every process in COPY(m) has crashed.
  - i.e. Q is dependent on m, but there is no way to replay m.
- Avoid orphans: for each non-stable message m, there is at most one process dependant on m.
  - The receiver of m, say process P, is not allowed to send any messages after the delivery of m without first written m to stable storage.