

## CSSE4004-Lecture 6

### Fault Tolerance

## Learning Objectives

After this week you should

- understand what are the basic concepts related to fault tolerance
- be able to describe failure models
- be able to describe approaches to achieve dependability of processes
- be able to describe approaches to achieve reliable communication
- understand how recovery can be achieved

2

## Basic Concepts

Dependability includes

- Availability
  - ready to use now
- Reliability
  - won't fail over time
- Safety
  - failure is not catastrophic
- Maintainability
  - failure easy to fix

3

## Why different in distributed systems?

- Partial failures possible in distributed systems
- Process distribution and process communication makes fault tolerance more difficult

4

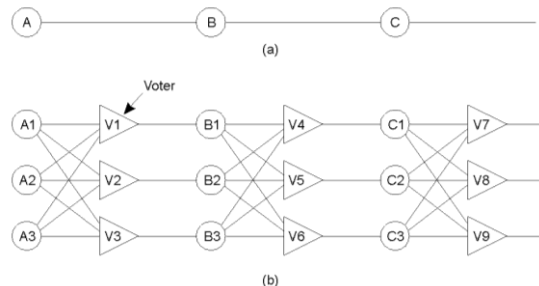
## Failure Models

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

Different types of failures

5

## Failure Masking by Redundancy



Triple modular redundancy

6

## Triple Modular Redundancy Rationale

Why 3 devices?

- > 1 failure unlikely
- > 1 failure *the same way* even less

Why 3 voters?

- They can also fail
- very general, widely (not universally) used*

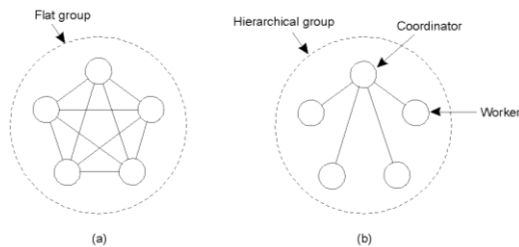
7

## Process Resilience

- Achieved through replication (process redundancy)
- Design issues
  - Flat vs. hierarchical groups
  - Agreement in faulty systems
    - Byzantine generals

8

## Flat Groups versus Hierarchical Groups



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

9

## Flat vs. Hierarchical Trade-Offs

Flat

- Symmetry
  - No 1 point of failure
- Complexity
  - voting

Hierarchical

- Coordinator fails, whole thing fails
- If coordinator OK, decisions simple

10

## Failure masking and replication...

Mask failures

- $k$  fault tolerant if can meet spec with  $k$  faulty components
  - fail silently:  $k + 1$  components enough
  - *Byzantine* failures (produce wrong answers):  $2k + 1$  components for  $k$  fault tolerance
- Assume processes don't team up to lie
- More complex if process group must *reach agreement*

11

## Agreement in Faulty Systems (1)

Possible cases:

1. Synchronous versus asynchronous systems
2. Communication delay is bounded or not
3. Message delivery is ordered or not
4. Message transmission is done through unicasting or multicasting

12

## Agreement in Faulty Systems (2)

		Message ordering				Communication delay
		Unordered		Ordered		
Process behavior	Synchronous			X		Bounded
	Asynchronous	X	X	X	X	Unbounded
		Unicast	Multicast	Unicast	Multicast	
		Message transmission				

Ta-St.8-4. Circumstances under which distributed agreement can be reached

13

## ...Agreement in Faulty Systems (3)

### Byzantine generals

- $n$  generals of whom  $m$  traitors (faulty)
  - each knows own troop strength
    - asks others by sending own correct strength
  - wants to know rest; at end vector:
    - element  $i$  = strength of general  $i$  if loyal, undefined otherwise
- Nice abstraction of real problems

14

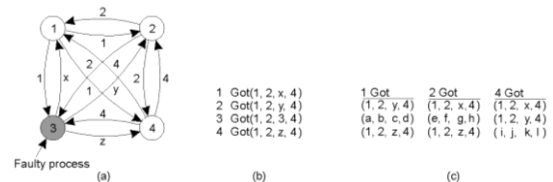
## ...Agreement in Faulty Systems (4)

Byzantine generals: Lamport's Algorithm

1. Every general sends reliable message to all others informing them of its troop strength
2. Collect results together at each general
3. Send results each general sees to each other general
4. Find majority vote for general  $i$  in final results

15

## ...Agreement in Faulty Systems (5)



Byzantine generals problem for 3 loyal generals, 1 traitor

- generals announce their troop strengths (units: kilosoldiers)
- vectors that each general assembles based on (a)
- vectors that each general receives in step 3

16

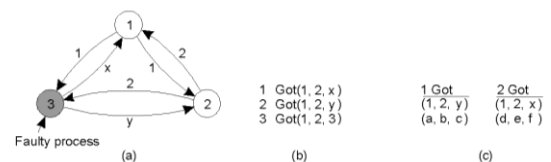
## ...Agreement in Faulty Systems (6)

Byzantine generals: Lamport principles

- For  $m$  faulty processes need  $2m+1$  correct processes
- Total number needed for up to  $m$  bad processes:  $3m+1$ 
  - overall effect: need  $> 2/3$  of processes correct

17

## ...Agreement in Faulty Systems (8)



As before, except 2 loyal generals, 1 traitor

18

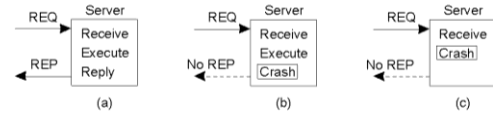
## Reliable Client-Server Communication

a lot handled by network ...

- RPC in presence of failures
  - client can't locate server
    - can't make RPC completely transparent
  - lost request messages
    - complexities: see *lost replies*
  - Server crashes
  - Lost reply messages
  - Client crashes

19

## Server Crashes...



A server in client-server communication

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

20

## ...Server Crashes...

Sequence of events: 6 different orders of

- Send completion message (M)
- Print text (P)
- Crash (C)

Sequence MC(P) means:

- Send completion message
- Crash
- Now can't print text, so "(P)" parenthesized

21

## ...Server Crashes

Reissue strategy	Strategy M -> P			Strategy P -> M		
	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
	Always	DUP	OK	OK	DUP	DUP
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 – none always OK.

22

## Lost Reply Messages

difficult: timeout  $\neq$  lost message generally

- *idempotent* request: can be repeated
- Is request to transfer money idempotent?
- Need sequence numbers
- Results have to be stored for some time

23

## Client Crashes

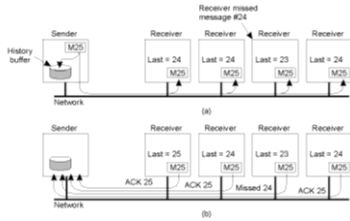
*orphan*: computation with no client

- problems
  - waste CPU
  - lock files, tie up resources
  - what if client restarts, reissues request?
- solutions
  - *extermination*: record RPCs, kill on reboot
  - *reincarnation*: broadcast new *epoch* on reboot: all remote computations restart
    - *gentle reincarnation*: only kill if owner not found
  - *expiration*: if not given new time quantum, die

*problems with all of these ...*

24

## Basic Reliable-Multicasting Schemes



A simple solution to reliable multicasting when all receivers are known and assumed not to fail

- Message transmission
- Reporting feedback

25

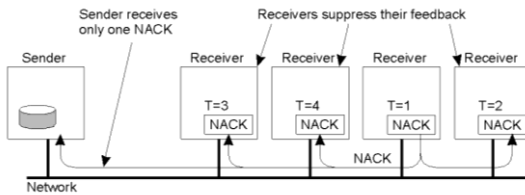
## Scalability in Reliable Multicasting

Problem:  $N$  receivers, 1 message  $\rightarrow N$  acks (acknowledgement or feedback *implosion*)

- only return feedback if message missed
  - scales better
  - can still get implosion
  - practical limit on how long message can be buffered at source
- solutions
  - hierarchical and non-hierarchical feedback control

26

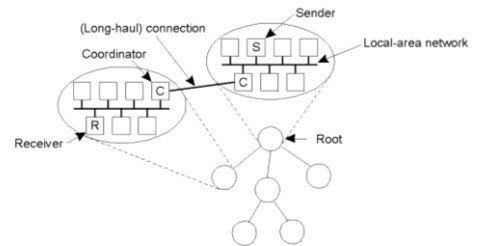
## Nonhierarchical Feedback Control



Several receivers scheduled request for retransmission, but 1st retransmission request leads to suppression of others

27

## Hierarchical Feedback Control



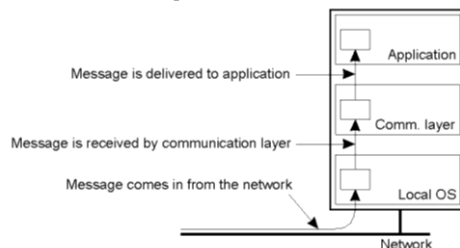
Essence of hierarchical reliable multicasting:

- Each local coordinator forwards message to its children
- Local coordinator handles retransmission requests

28

## Atomic multicast -Virtual Synchrony...

*virtual synchronous* multicast: sender crashes, deliver to all rest of processes or none

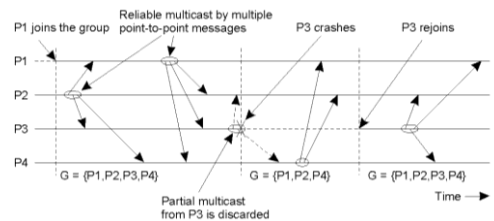


logical organization of distributed system to distinguish between message receipt and message delivery

29

## ...Virtual Synchrony

*virtual synchronous* multicast: sender crashes, deliver to all rest of processes or none



30

## Message Ordering...

Multicasts within epochs defined by view changes  
(group membership changes)

What of ordering within an epoch?

- unordered
  - no guarantee of order processes see messages
- FIFO-ordered
  - messages from same process seen in same order
- Causally-ordered
  - even if not from same sender causal ordering maintained
- Totally-ordered
  - delivered in same order to all group members (in addition to any of the above)

31

## ...Message Ordering...

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

3 communicating processes in same group –  
ordering of events per process along vertical axis

*unordered delivery*

32

## ...Message Ordering

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	

4 processes in same group with 2 different senders, and  
possible delivery order of messages under FIFO-  
ordered multicasting

*total-ordered delivery*: deliver to all group members in  
*same order*

33

## Implementing Virtual Synchrony ...

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

6 versions of virtually synchronous reliable multicasting

*atomic multicasting*: virtually synchronous reliable  
multicasting with totally-ordered delivery

34

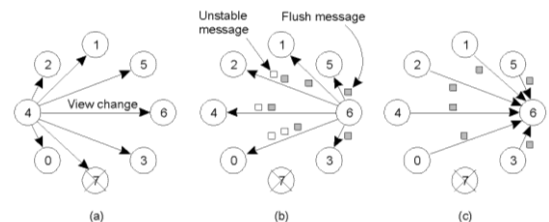
## ...Implementing Virtual Synchrony...

Based on Isis – uses reliable TCP point-to-point communication

- The problem:
  - make sure all non-crashed processes receive all messages before processing view change
- TCP makes possible to ensure FIFO ordering
  - no guarantee that sender sent all copies before crashing
- *stable* message
  - received by all group members
- On *view change*
  - processes send copy of any unstable message to rest of group
    - duplicates not a problem
  - issue *flush* message for new view
- Some complications to get everything right...

35

## ...Implementing Virtual Synchrony



- Process 4 notices process 7 has crashed, sends view change
- Process 6 sends out all its unstable messages, followed by flush message
- Process 6 installs new view when received flush message from everyone else

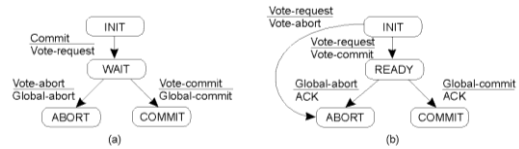
36

## Distributed commit

- Atomic multicasting is an example of a more general problem:
  - distributed commit*
- Distributed commit requires that an operation is performed by each member of a process group or none at all
- Two- and three-phase commit protocols

37

## Two-Phase Commit...



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

38

## ...Two-Phase Commit...

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...

### 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 steps by coordinator in a 2PC protocol

40

## ...Two-Phase Commit...

steps 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;
}
    
```

41

## ...Two-Phase Commit

### 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 for handling incoming decision requests

42

## Three-Phase Commit (1)

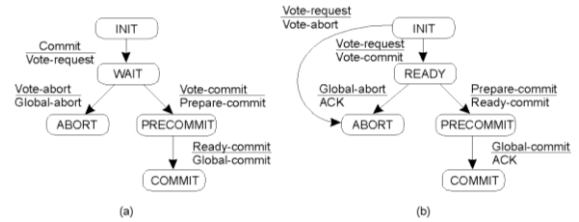
The states of the coordinator and each participant satisfy the following two conditions:

1. There is no single state from which it is possible to make a transition directly to either a COMMIT or an ABORT state
2. There is no state in which it is not possible to make a final decision, and from which a transition to a COMMIT state can be made

43

## Three-Phase Commit (2)

2PC problem: blocked while coordinator crashed and down



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

44

## 3PC vs. 2PC

conditions of failure of 2PC rare

- worth studying details of 3PC
  - useful example to understand issues
- in practice
  - 2PC usually considered sufficient

45

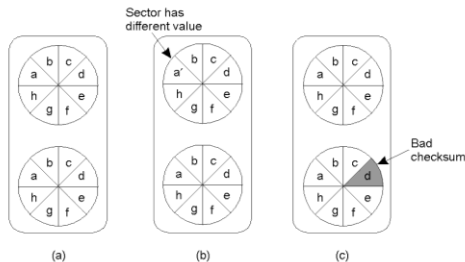
## Recovery

*recovery from error* fundamental to fault tolerance

- backward recovery
  - back from erroneous state to previous correct state
  - network example: retransmission if packet lost
- forward recovery
  - construct correct state from error state
  - network example: *erasure correction*
- implementation details
  - stable storage
  - checkpointing
  - message logging

46

## Recovery Stable Storage



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

47

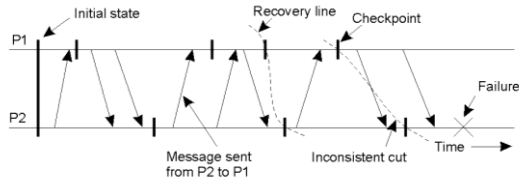
## Checkpointing...

*distributed snapshot*: consistent global state

- recovery line
  - most recent distributed snapshot
  - local states in stable storage used to recover global state
- problem:
  - domino effect from *independent checkpointing*
- solution
  - record dependences to allow consistent state to be found

48

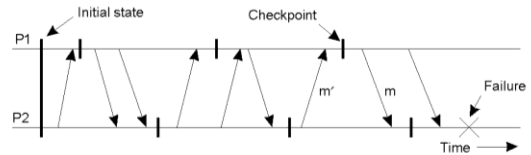
## Checkpointing



A recovery line

49

## Independent Checkpointing

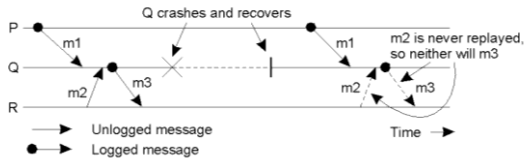


domino effect – could roll back to initial state

50

## Message Logging

- can reduce number of checkpoints



incorrect replay of messages after recovery,  
leading to an orphan process

51

## Message-Logging Schemes

assume messages have sender, receiver, sequence number  
Stable message: can no longer be lost (e.g. in stable storage)

- Pessimistic logging
  - each nonstable message delivered to at most one process
  - avoids orphan processes
  - main work *before* a crash
- Optimistic logging
  - roll each process back to state where does not depend on delivery of message some process(es) do not yet have in stable storage
  - main work *after* crash
- Generally
  - pessimistic easier to get right, more practical

52

## Summary

how differs from other fault tolerance, networking

- Handling failure hard
  - hard even to know it occurred (e.g., Byzantine failures)
- Supporting redundancy hard
  - multicasting needs to be *reliable*
  - scalability a key issue
  - atomicity hard
- Recovery hard
  - distributed checkpointing hard
- Generally
  - same issues as for nondistributed
  - networks + fault tolerance multiplies difficulty
- Reading: Chapter 8

53