

Today: Fault Tolerance

- Failure models
- Agreement in presence of faults
 - Two army problem
 - Byzantine generals problem
- Reliable communication
- Distributed commit
 - Two phase commit
 - Three phase commit
- Failure recovery
 - Checkpointing
 - Message logging



Replica Management

- Replica server placement
 - Web: geographically skewed request patterns
 - Where to place a proxy?
 - K-clusters algorithm
- Permanent replicas versus temporary
 - Mirroring: all replicas mirror the same content
 - Proxy server: on demand replication
- Server-initiated versus client-initiated



Content Distribution

- Will come back to this in Chap 12
- CDN: network of proxy servers
- Caching:
 - update versus invalidate
 - Push versus pull-based approaches
 - Stateful versus stateless
- Web caching: what semantics to provide?

Final Thoughts

- Replication and caching improve performance in distributed systems
- Consistency of replicated data is crucial
- Many consistency semantics (models) possible
 - Need to pick appropriate model depending on the application
 - Example: web caching: weak consistency is OK since humans are tolerant to stale information (can reload browser)
 - Implementation overheads and complexity grows if stronger guarantees are desired

Fault Tolerance

- 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
 - $\text{Prob}(\text{failure}) = \text{Prob}(\text{Any one component fails}) = 1 - P(\text{no failure})$



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

- 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



Basic Concepts (contd)

- Fault tolerance: system should provide services despite faults
 - Transient faults
 - Intermittent faults
 - Permanent faults

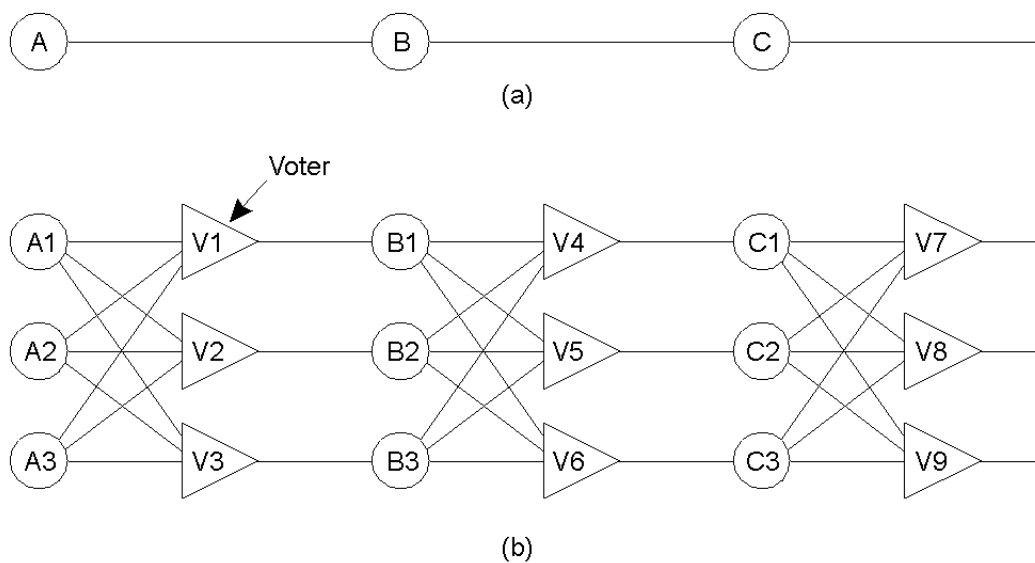


Failure Models

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure <i>Receive omission</i> <i>Send omission</i>	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 <i>Value failure</i> <i>State transition failure</i>	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.

Failure Masking by Redundancy



- Triple modular redundancy.

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

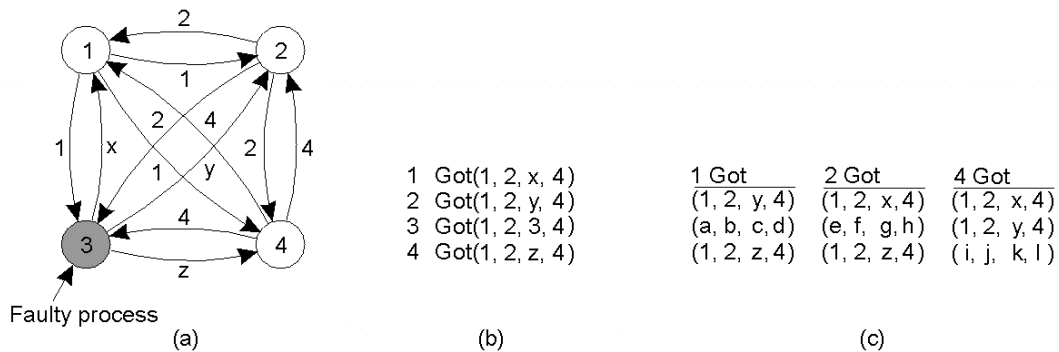


Byzantine Faults

- 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



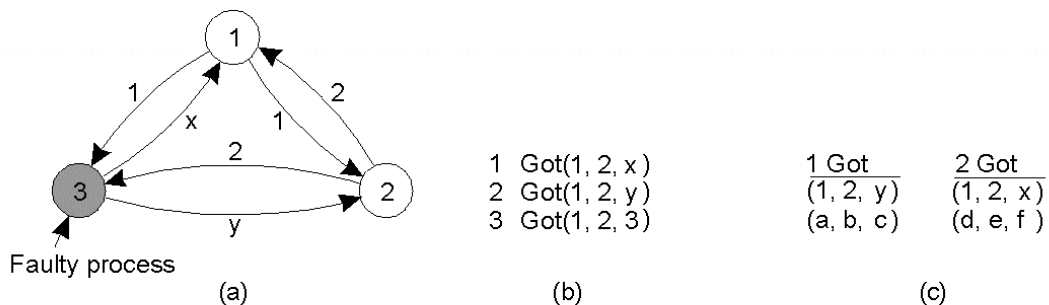
Byzantine Generals Problem



- 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



- 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 $2m+1$ processes function correctly [Lamport 82]
 - Need more than two-thirds processes to function correctly

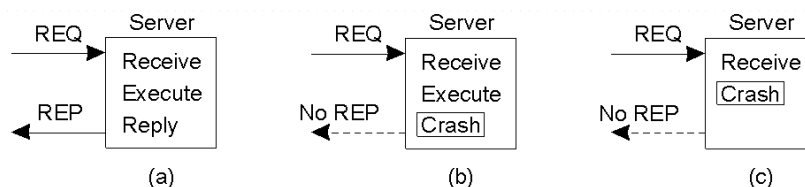


Reaching Agreement

- If message delivery is unbounded,
 - No agreement can be reached even if one process fails
 - Slow process indistinguishable from a faulty one
- BAR Fault Tolerance
 - Until now: nodes are byzantine or collaborative
 - New model: Byzantine, Altruistic and Rational
 - Rational nodes: report timeouts etc

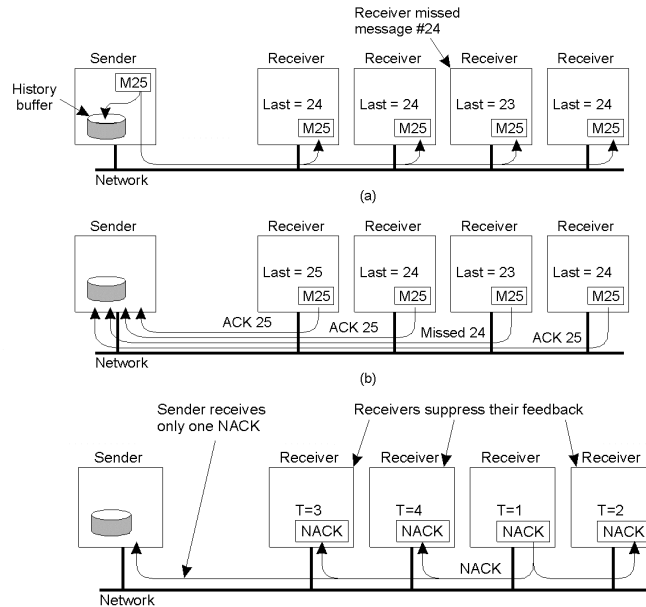
Reliable One-One Communication

- 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



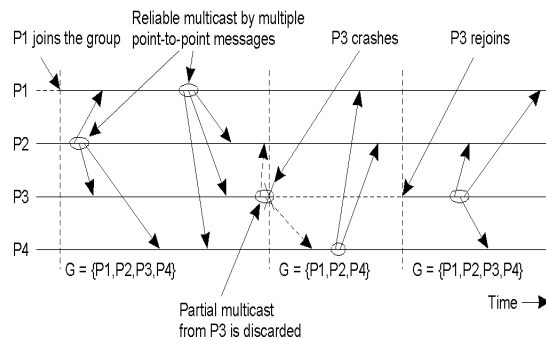
Reliable One-Many Communication

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



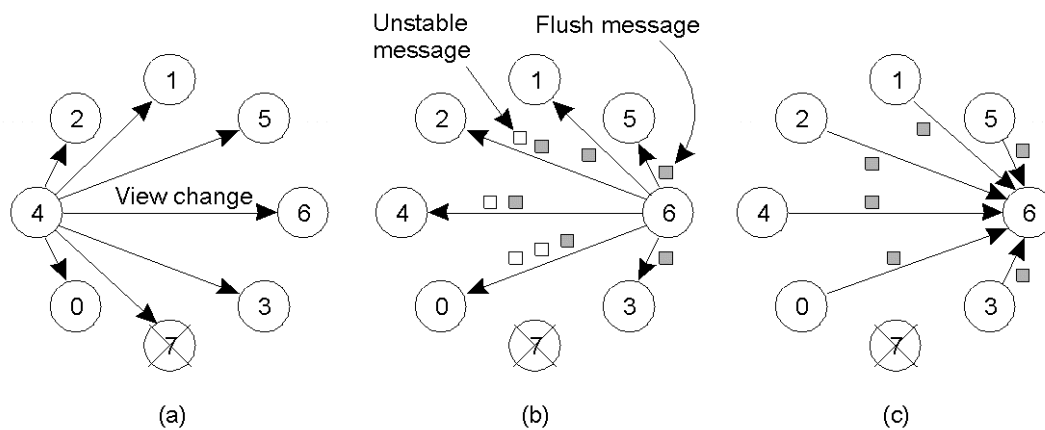
Atomic Multicast

- 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



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



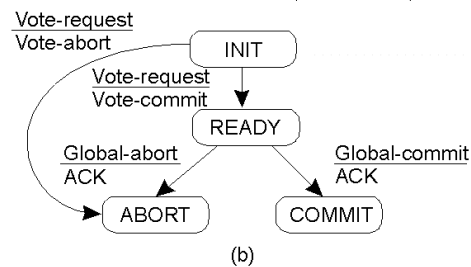
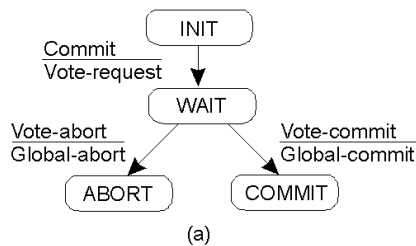
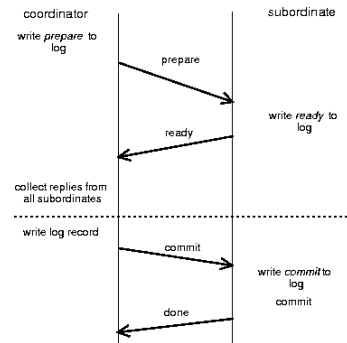
Distributed Commit

- 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 */

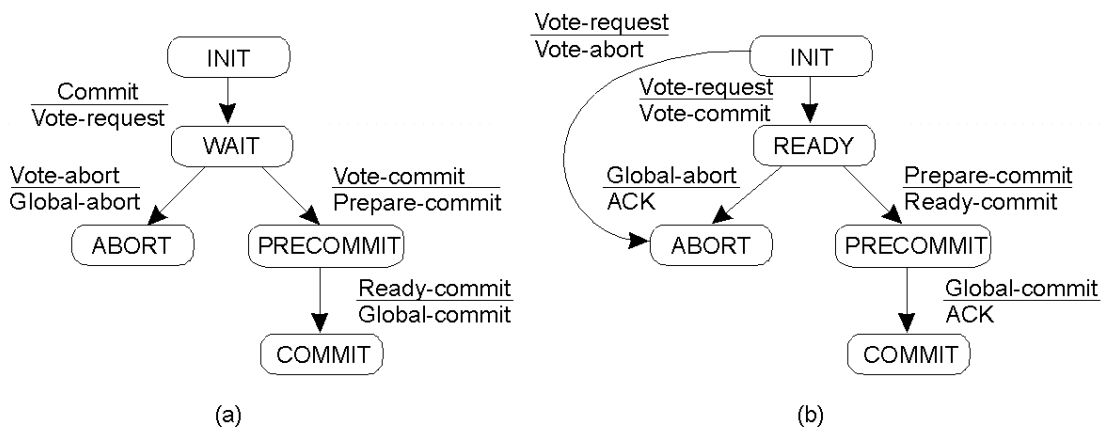
```

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



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

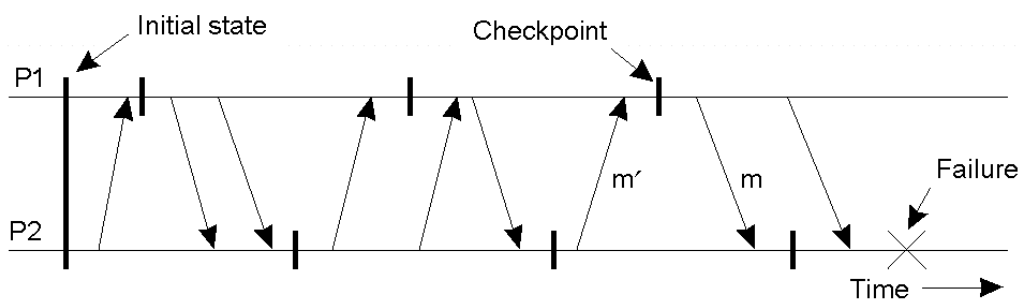
Three phase commit: variant of 2PC that avoids blocking



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 [discussed in Lec 11]
- 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