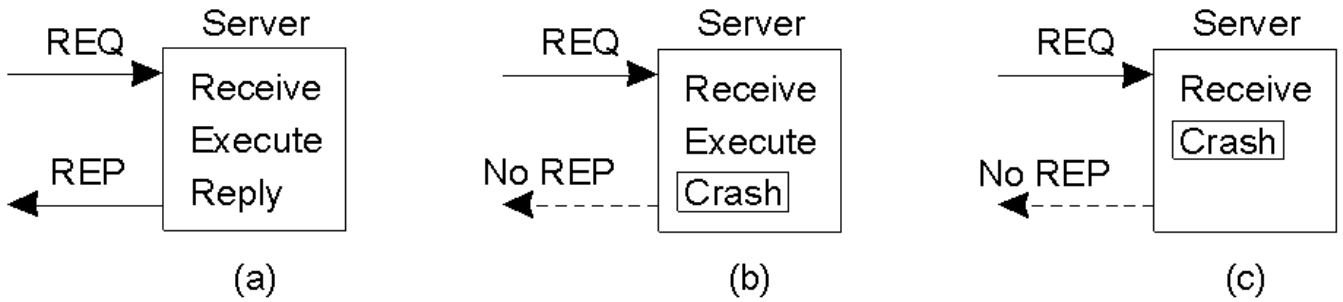
Today: Fault Tolerance

- Reliable communication
- Distributed commit
  - Two phase commit
  - Three phase commit
- Paxos
- Failure recovery
  - Checkpointing
  - Message logging
- Byzantine faults

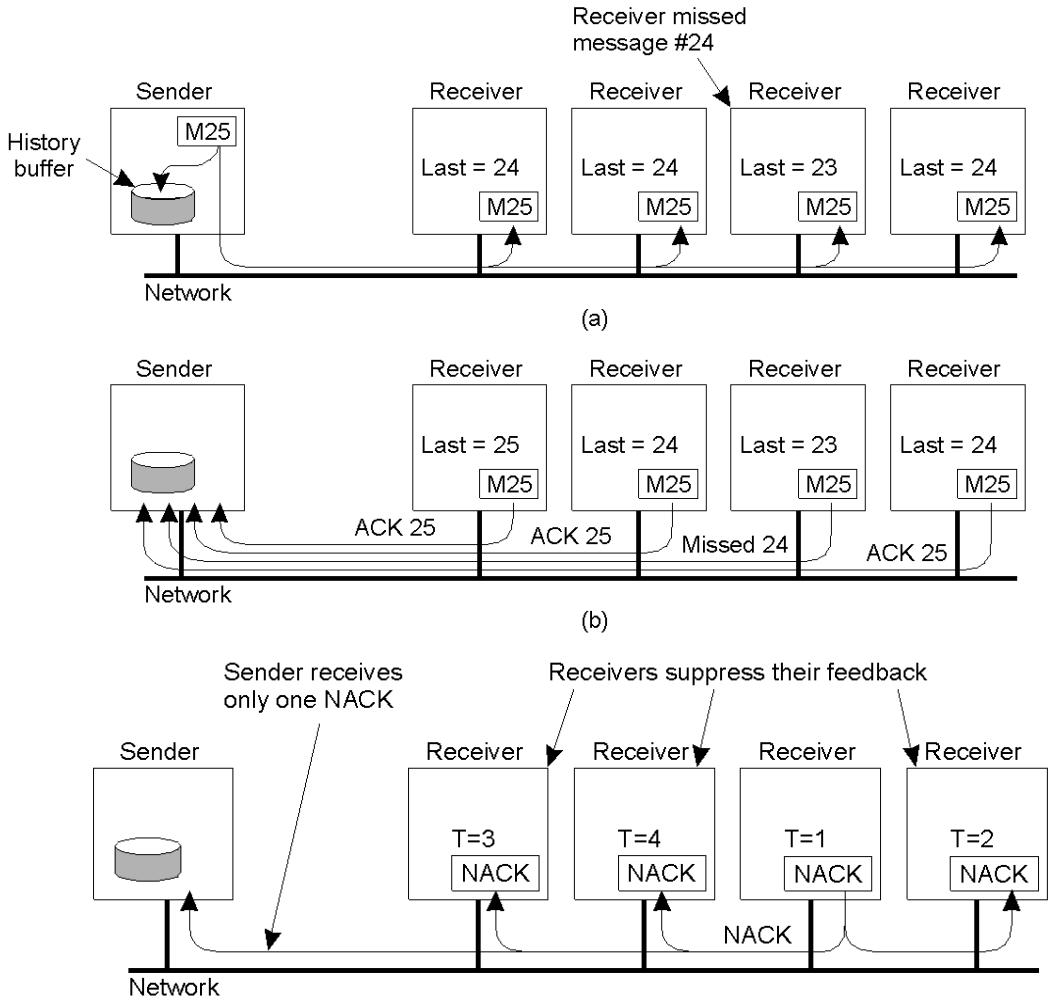
Reliable One-One Communication

- 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



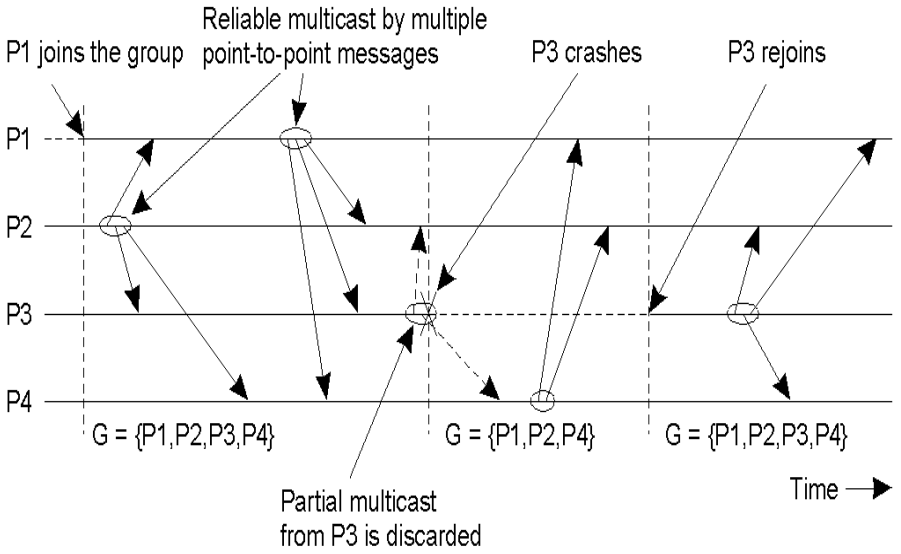
Reliable One-Many Communication

- Lost messages => need to retransmit
- Possibilities
  - ACK-based schemes
    - Sender can become bottleneck
  - NACK-based schemes
- Q: Can messages still get lost?



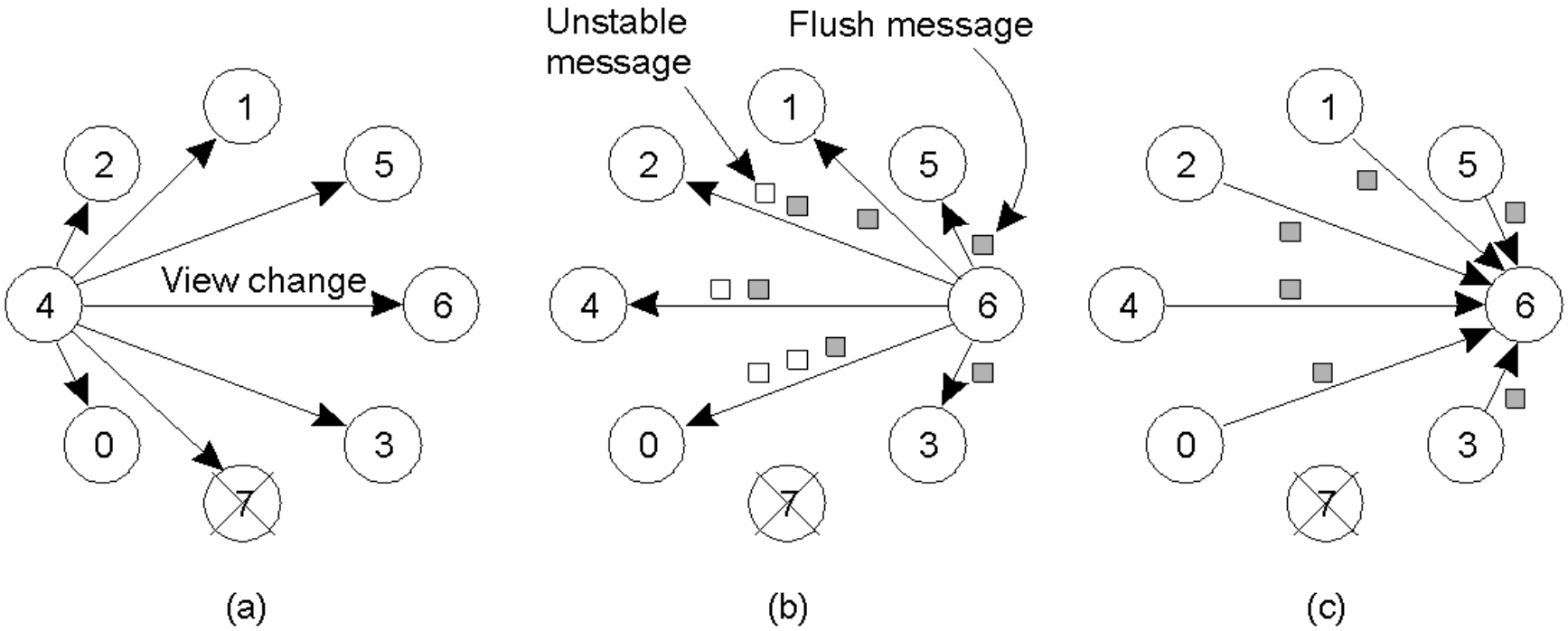
Reliable Multicast

- *Reliable multicast:* all processes receive the message or none at all
  - Example: replicated database
- Problem: how to handle process crashes?
  - Updates can be missed
- Solution: Re-broadcasts
  - *f* processes re-broadcast the message
  - After receiving *f* confirmations, message is stable
  - Q: Can something still go wrong?
- Solution: *Group views*
  - View = set of processes + initial history
  - Agreement to move to the next new



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

Variants of Multicast

- *Reliable multicast:* all processes receive the message or none at all
  - All-or-nothing is typically called atomicity but the term is overloaded so the literature uses the term *reliable* here
- Ordering requirements are orthogonal
  - *Atomic multicast:* reliable multicast + total order on message delivery
  - *FIFO, causal order* are orthogonal to both total order and reliability

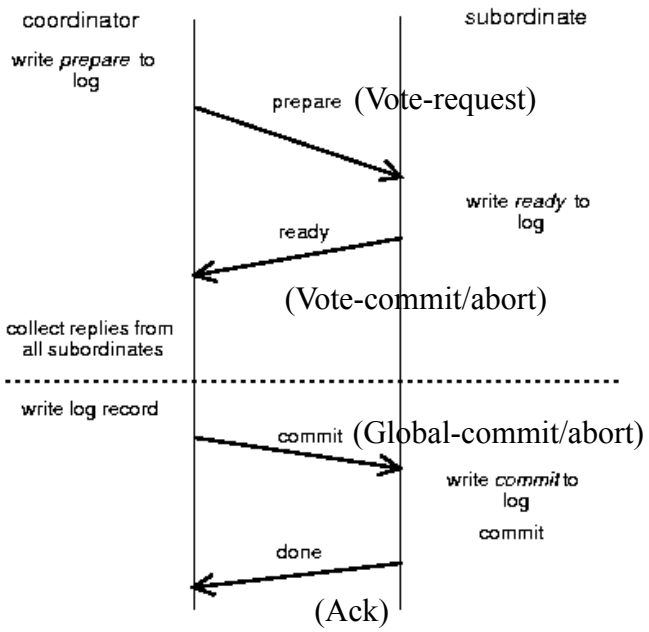
Multicast	Basic Message Ordering	Total-Ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

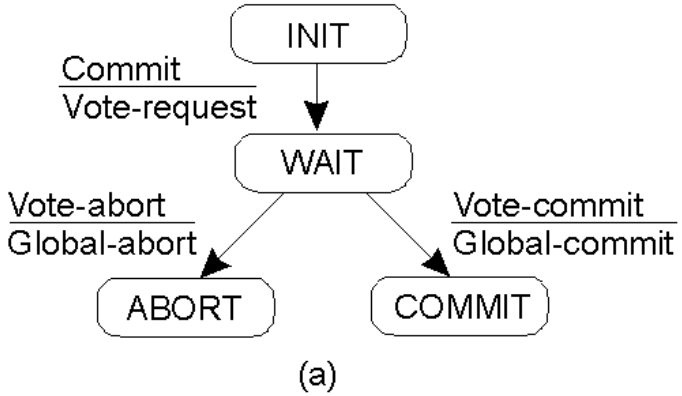
Distributed Commit

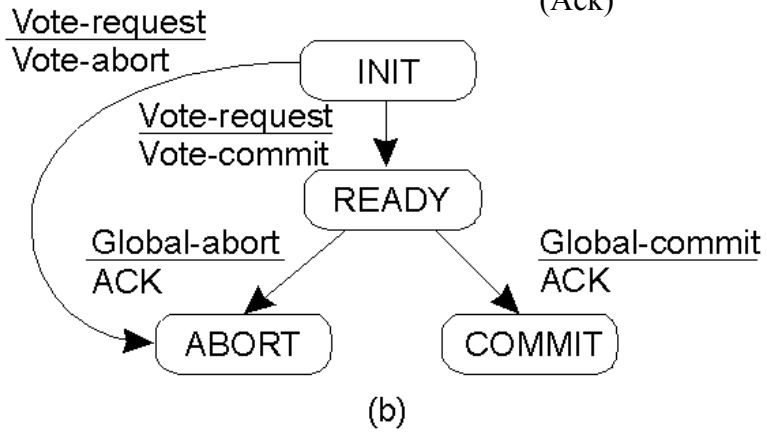
- Atomic multicast example of a more general problem
  - All processes in a group perform an operation or not at all
  - Examples:
    - Atomic multicast: Operation = delivery of a message in order
    - Distributed transaction: Operation = commit transaction
- Problem of distributed commit
  - Atomic execution of transactions in a group of processes
- Possible approaches
  - Two phase commit (2PC) [Gray 1978]
  - Three phase commit

Two Phase Commit

- Executed at the end of a transaction
- Coordinator + participants
- Crash/recovery fault model
- Involves two phases
  - Voting phase: processes vote on whether to commit
  - Decision phase: actually commit or abort







Coordinator

Participant

Implementing Two-Phase Commit

actions by coordinator:

```
write INIT to local log;

multicast VOTE_REQUEST to all participants;

while not all votes have been collected {
    wait for any incoming vote or timeout;
    if timeout {
        write GLOBAL_ABORT to local log;
        multicast  GLOBAL_ABORT to all participants;
        exit;
    } else record vote;
}
if all participants sent VOTE_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;
}
```

Implementing 2PC

actions by participant:

```
write INIT to local log;
wait for VOTE_REQUEST from coordinator or timeout;
if timeout {
    write VOTE_ABORT to local log;
    exit;
}

if participant votes COMMIT {
    write READY 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; /* remains 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 ABORT to local log;
    send  VOTE ABORT to coordinator;
}
```

actions for handling decision requests: /*executed by separate thread */

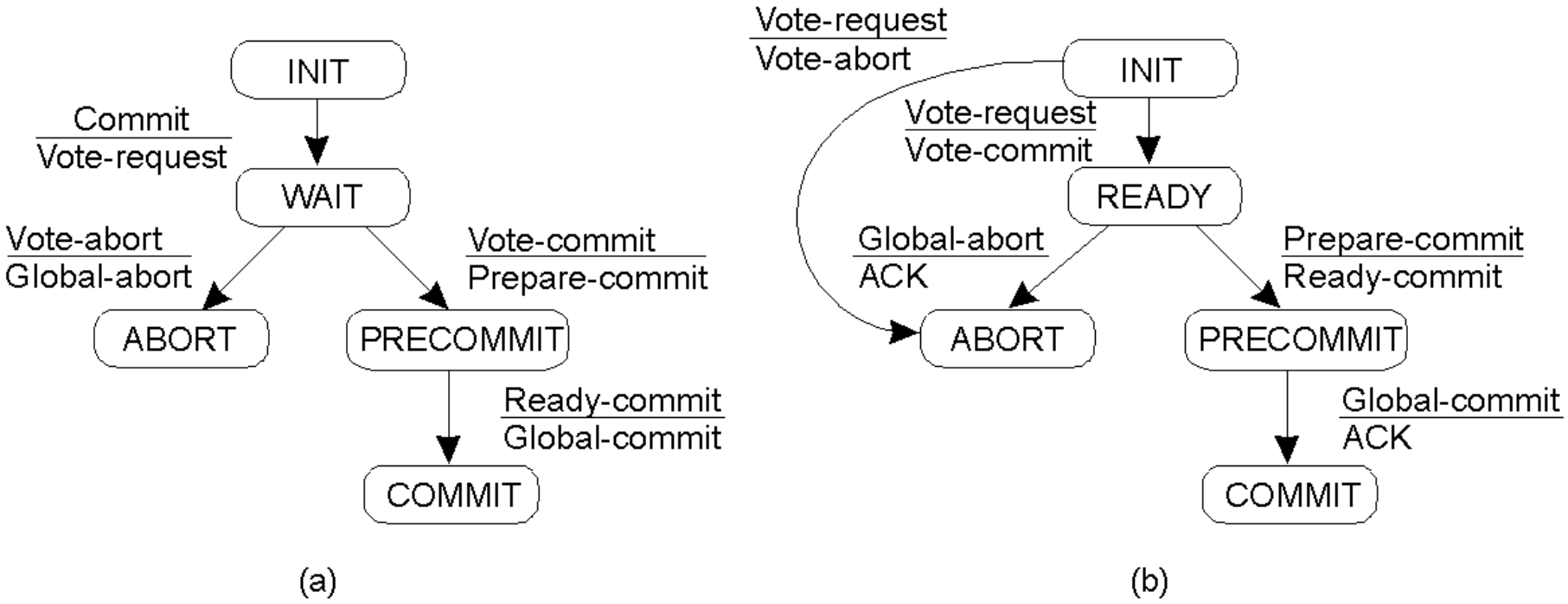
```
while true {

    wait for incoming DECISION_REQUEST

    /*blocked */

    read current 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 */
```

Recovering from a Crash

- If P in READY, needs to contact another process (*Q:* why only then?)
- Process P contacts process Q if coordinator crashes
  - If INIT : abort locally and inform coordinator
  - If READY: contact another process Q and examine Q's state
- *Q:* When does the protocol block?

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

Three-Phase Commit



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

Three phase commit: variant of 2PC that avoids blocking

Consensus, Agreement

- Consensus: formalization of distributed agreement
  - Requires processes to agree on data value needed for computation
  - Examples: whether to commit a transaction, agree on identity of a leader, atomic broadcasts, distributed locks
- Properties of a consensus protocol with fail-stop failures
  - *Agreement:* every (correct) process agrees on same value
  - *Termination:* every correct process decides some value
  - *Validity:* If all propose v, all (correct) processes decide v
  - *Integrity:* Every correct process decides at most one value and if it decides v, someone must have proposed v.

Safety vs. Liveness

- Safety: *never* reach an incorrect output
  - Both 2PC and 3PC are safe: it never happens that one participant commits and another aborts
- Liveness: *eventually* all correct participants terminate
  - 2PC: Must wait for all nodes and coordinator to be up
  - 3PC: Network partitions are still an issue
  - Liveness guaranteed only in *good runs*
- Fundamental impossibility result by Fischer, Lynch, Patterson (aka FLP result)

  "It is impossible to reach distributed consensus in an asynchronous system with one faulty process"

  - This is because it is impossible to distinguish a faulty process from a slow one without making assumptions about timeouts (e.g. accurate fault detection)