Distributed Systems
COMP 212

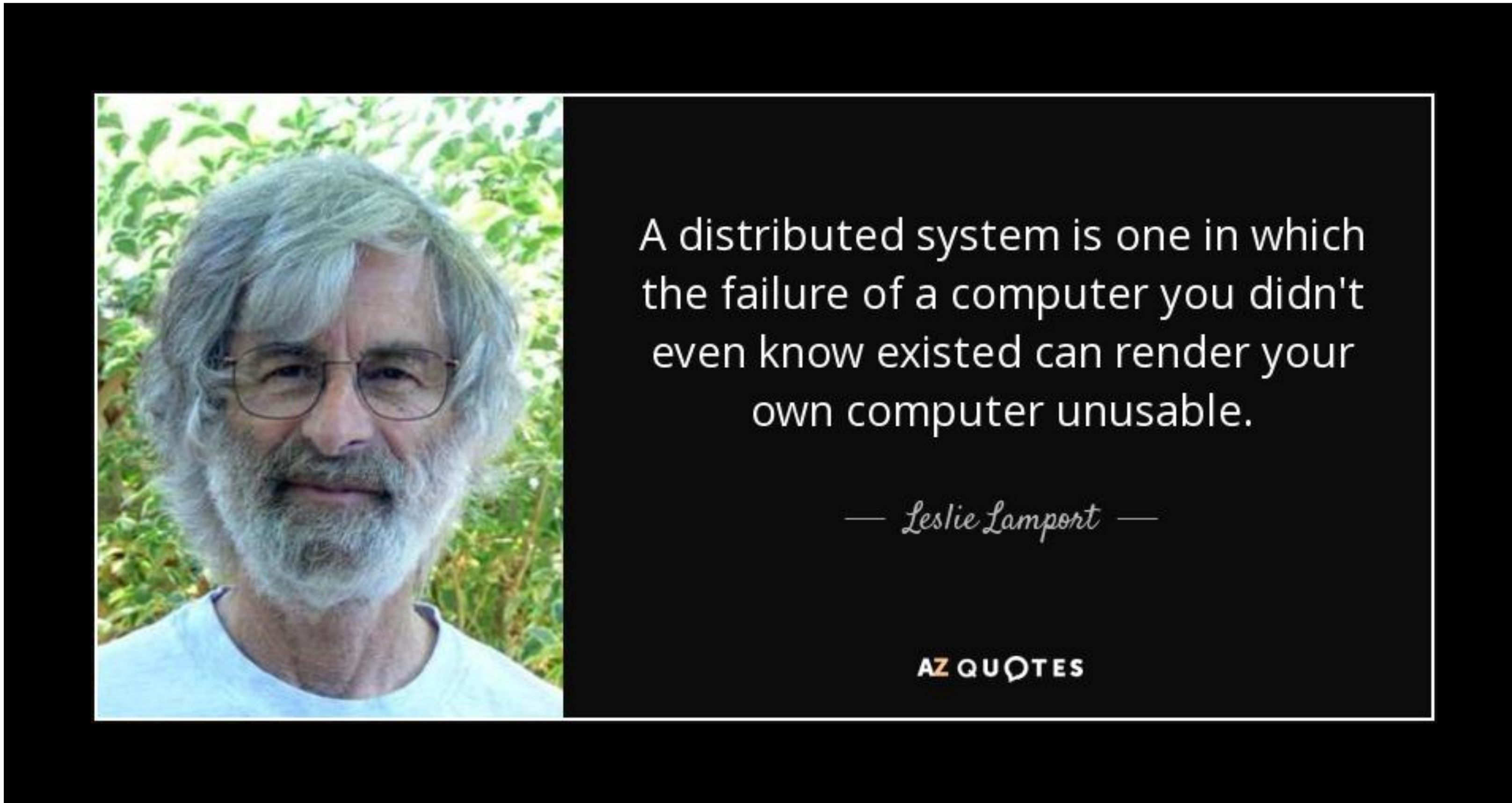
Lecture 19

Othon Michail

UNIVERSITY OF LIVERPOOL

Fault Tolerance

What is a Distributed System?



Distributed vs Single-machine Systems

- A key difference: partial failures
  - One component fails
  - Some components are affected
  - Others not
- In single-machine systems failures are often total
  - The whole system is affected

- Important and challenging goal:

Design Distributed Systems that can mask partial failures and automatically recover from them without seriously affecting the overall performance

- This is the concept of fault tolerance

Basic Concepts

- Fault Tolerance is closely related to the notion of Dependability

For a DS to be dependable, it should satisfy:

- **Availability**: the system is ready to be used immediately
- **Reliability**: the system can run continuously without failure
- **Safety**: if a system fails, nothing catastrophic will happen
- **Maintainability**: when a system fails, it can be repaired easily and quickly (and, sometimes, without its users noticing the failure)
- **Security**: protection of a system from various threats

But, what is a “Failure”?

- A definition:
  - A system is said to fail when it *cannot meet* its promises
- Failures are caused by errors in the system
  - a damaged packet
  - a lost packet
  - a process not responding
  - a server sending unexpected responses
- Errors are caused by faults
  - We often detect an error but cannot easily say what caused it
  - a bad transmission medium causing damage to packets
  - an overheated CPU making a server behave in unexpected ways
- *A fault tolerant system can provide its services even in the presence of faults*

Main Types of Faults

- **Transient** fault: occurs once and then disappears
  - A bird flying through a beam of a microwave transmitter
  - Some bits might get lost but a retransmission will probably work

- **Intermittent** fault: may reappear again and again
  - A loose contact on a connector

- **Permanent** fault: continues to exist until the faulty component is replaced
  - burn-out chips, software bugs, disk head crashes

Main Types of Failures

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 outgoing 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 (aka **Byzantine**) failure	A server may produce arbitrary responses at arbitrary times (even malicious)

Failure Masking by Redundancy

- Strategy: if we cannot avoid failures then better hide them from other processes and/or users
  - using redundancy

Three main types:

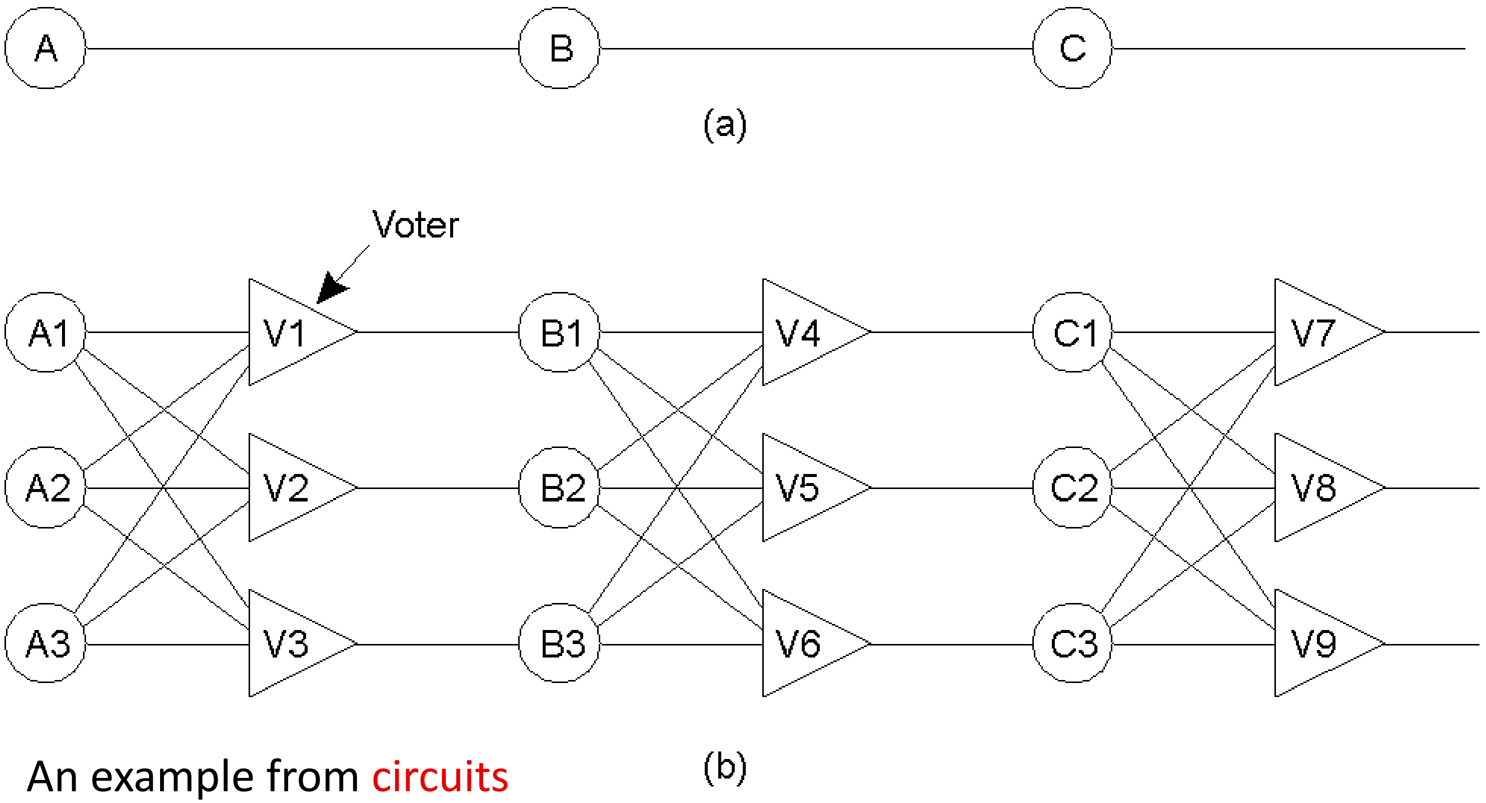
1. Information Redundancy
   - Add extra bits to allow for error detection/recovery
   - e.g., parity bits, Hamming codes

2. Time Redundancy
   - Perform operation and, if required, perform it again
   - Well suited for transient and intermittent faults

3. Physical Redundancy
   - Add extra (duplicate) hardware and/or software components to the system
   - Replication

Example: Physical Redundancy

- Ubiquitous technique: nature (two eyes, ears, lungs), aircraft engines, sports referees



- An example from circuits
- Triple modular redundancy: Each device is replicated three times. If two or three inputs of a voter are the same, the output is equal to that input (majority wins). Note that a voter itself might be faulty, hence, a separate voter at each stage.

DS Fault Tolerance Topics

- Process Resilience
  - Process failure prevention by replicating processes into groups
    - Design issues
    - How to achieve agreement within a group when not all members can be trusted?
- Reliable Client/Server Communications
  - masking crash and omission failures
- Reliable Group Communication
  - e.g., what happens if a process joins the group during communication?
- Distributed COMMIT
  - operation to be performed by all group members, or none at all
- Recovery Strategies
  - Recovery from an error is fundamental to fault tolerance

Process Resilience

- Key approach to tolerating a faulty process:

  Organise several identical processes into a group.

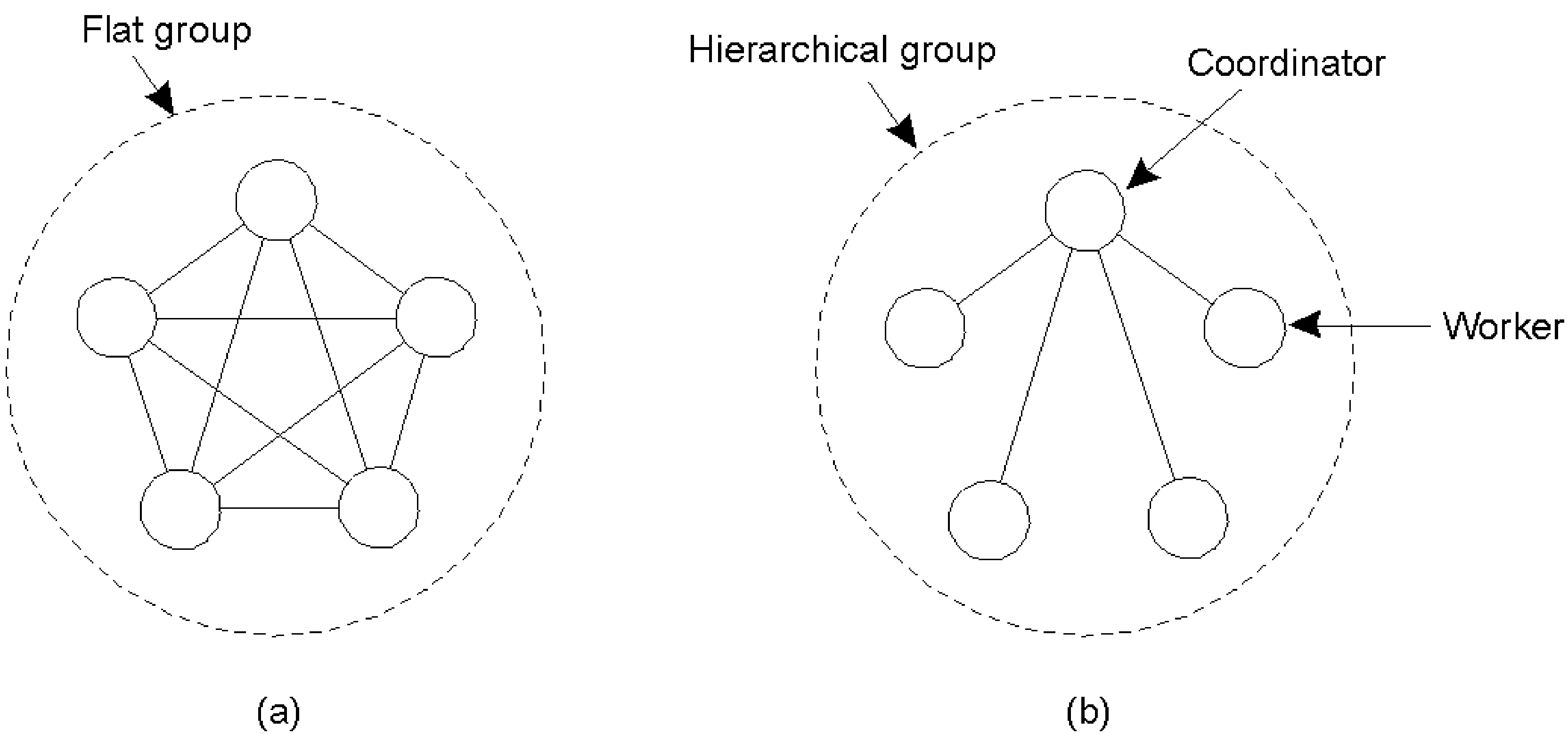
- Key property of groups:

  A message sent to the group is received by *all* "copies" of the process (the group members).

- If one process fails, hopefully some other process will still be able to function (and service any pending request or operation)

- Like sending an email to the email address of a department and then that email being automatically forwarded to a number of people so that the one available will handle it

Flat vs. Hierarchical Groups



a) **Communication in a flat group**: all the processes are equal, decisions are made collectively
   - Note: no single point-of-failure
   - However: decision making is complicated as consensus is required

b) **Communication in a simple hierarchical group**: one of the processes is elected to be the coordinator, which selects another process (a worker) to perform the operation
   - Note: single point-of failure
   - However: decisions are easily and quickly made by the coordinator without first having to ensure consensus

Reliable Client/Server Comms.

- In addition to process failures, a communication channel may exhibit crash, omission, timing, and/or arbitrary failures
- In practice, the focus is on masking *crash* and *omission* failures
- ***For example***: the point-to-point TCP masks omission failures by guarding against lost messages using ACKs and retransmissions. However, it performs poorly when a crash occurs (although a DS may try to mask a TCP crash by automatically re-establishing the lost connection).

Reliable Group Communication

- Reliable multicast services guarantee that all messages are delivered to all members of a process group

- Sounds simple, but is surprisingly tricky
  - as multicasting services tend to be inherently unreliable

- For a small group, multiple, reliable point-to-point channels will do the job, however, such a solution scales poorly as the group membership grows

Also:

- What happens if a process joins the group during communication?

- Worse: what happens if the sender of the multiple, reliable point-to-point channels crashes half way through sending the messages?

What is Reliable Group Communication?

- Processes may fail
  - All non-faulty members receive the message
  - Agree what the group looks like *before* a message can be delivered + other constraints
- All processes operate correctly
  - Message should be delivered to every current group member
  - Simplest case:
    - No process join or leave the group

Atomic Multicasting

- There often exists a requirement where the system needs to ensure that all processes get the message, or that none of them gets it

- An additional requirement is that all messages arrive at all processes in sequential order

- This is known as the "atomic multicast problem"

Example

- Consider a replicated database constructed on top of a distributed system
- DS offers reliable multicasting
  - DB is a *group of processes* one process per replica
- Suppose a replica crashes and then recovers
  - Some updates may be missed
  - Bringing the replica back requires knowing missed updates and their order (costly)
- With atomic multicasting:
  - Updates are only sent to “alive” processes
  - Before the crashed replica joins the group, it has to explicitly synchronise

Distributed COMMIT

General Goal:

- We want an operation to be performed by all group members, or none at all

Atomic multicasting:

- The operation is the delivery of the message

- There are three types of a “commit protocol”:
  - single-phase commit
  - two-phase commit
  - three-phase commit

The Two-Phase Commit Protocol

Summarised: GET READY, OK, GO AHEAD

1. The coordinator sends a *VOTE_REQUEST* message to all group members
2. The group member returns *VOTE_COMMIT* if it can commit locally, otherwise *VOTE_ABORT*
3. All votes are collected by the coordinator. A *GLOBAL_COMMIT* is sent if all the group members voted to commit. If one group member voted to abort, a *GLOBAL_ABORT* is sent.
4. The group members then **COMMIT** or **ABORT** based on the last message received from the coordinator

Problems with Failures

- Participants wait for messages. Crash?
  - Timeouts are used
  - e.g., if some participants do not reply to *VOTE_REQUEST* within a time limit, the coordinator will send *GLOBAL_ABORT* to all participants
  - What if a participant, *P*, sent *VOTE_COMMIT*, but did not get a message from the coordinator within the limit?
    - Coordinator crashed after asking P for a vote
    - Either immediately, or while asking others, or in process of informing other participants about the global decision…
    - Cannot just abort!

Blocking in 2 Phase Commit

- Either wait till the coordinator recovers (and inform again about the global decision)
- Or try and get a decision from a neighbour
  - If another process, *Q*, was not asked to vote, **ABORT**
  - If *Q* committed, **COMMIT**
  - But what if *Q* also waits for the final decision?

  - If the coordinator crashed just after asking everyone for votes, cannot reach a decision before the coordinator recovers…

Recovery Strategies

- Once a failure has occurred, it is essential that the process where the failure happened *recovers* to a correct state

- Recovery from an error is *fundamental* to fault tolerance

Two main forms of recovery:

1. **Backward Recovery**: return the system to some previous correct state (using *checkpoints*), then continue executing

2. **Forward Recovery**: bring the system into a correct state, from which it can then continue to execute

Forward and Backward Recovery (1)

Disadvantage of Forward Recovery:

- In order to work, all potential errors need to be accounted for up-front

- When an error occurs, the recovery mechanism then knows what to do to bring the system forward to a correct state

Forward and Backward Recovery (2)

Disadvantages of Backward Recovery:

- Checkpointing (can be very expensive, especially when errors are very rare)
- No guarantee that we won't meet the same error again
- Some operations cannot be rolled back

- [Despite the cost, backward recovery is implemented more often]

Recovery Example

Consider as an example: Reliable Communications

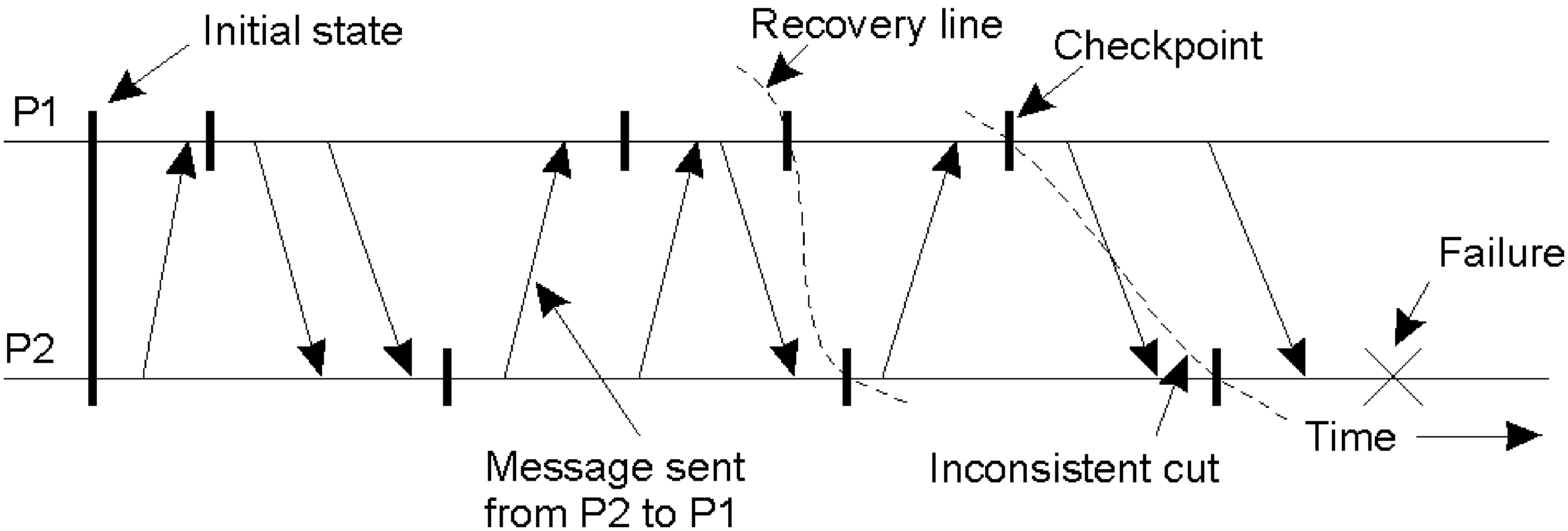
- Retransmission of a lost/damaged packet is an example of a backward recovery technique
- Recovery of damaged packets from others (e.g., consider Hamming codes) is an example of a forward recovery technique

Backward Recovery and Logging

- Many fault-tolerant DS combine checkpointing with logging
- Sender-based logs
  - Log a message before sending
- Receiver-based logs
  - Log a message before “executing”

Checkpointing

- Independent checkpointing



- May lead to inconsistencies and cascaded rollback (domino effect)
  - P2 has received *m* but P1 does not have a record of sending
  - Rollback to an earlier state

Coordinated Checkpointing

- All processes synchronise to jointly write their state to a local stable storage—no cascaded rollbacks
- Algorithms
  - Two-phase blocking protocol
    - Coordinator multicasts CHECKPOINT_REQUEST
    - When a process receives this message
      - It takes a local checkpoint
      - Stops accepting incoming messages
      - Queues *outgoing* messages handed by application
      - Acknowledges the coordinator
    - When everybody acknowledged, coordinator sends CHECKPOINT_DONE (unblocking processes)

Summary (1 of 2)

- Fault Tolerance:

The characteristic by which a system can mask the occurrence and recovery from failures. A system is fault tolerant if it can continue to operate even in the presence of failures.

- Types of failures:
  - *Crash* (system halts)
  - *Omission* (incoming request ignored)
  - *Timing* (responding too soon or too late)
  - *Response* (getting the order wrong)
  - *Arbitrary/Byzantine* (indeterminate, unpredictable)

Summary (2 of 2)

- Fault Tolerance is generally achieved through use of *redundancy* and *reliable multicasting protocols*
- Processes, client/server and group communications can all be "enhanced" to tolerate faults in a distributed system. Commit protocols allow for fault tolerant multicasting (with *two-phase* the most popular type).
- *Recovery* from errors within a Distributed System tends to rely heavily on **Backward Recovery** techniques that employ some type of *checkpointing* or *logging* mechanism, although **Forward Recovery** is also possible