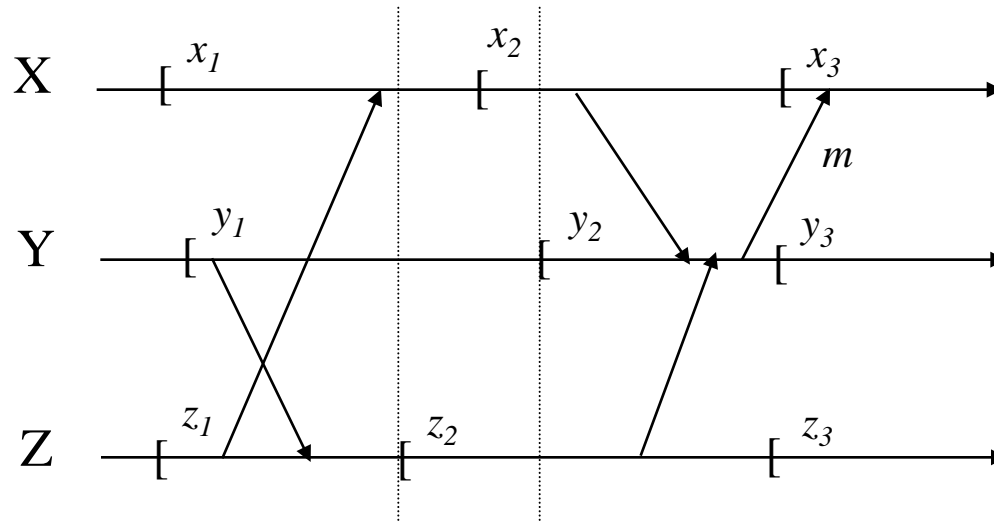


# Consistent set of checkpoints

- $\{x_2, y_2, z_2\}$  is a strongly consistent set of checkpoints



# Strongly/simple consistent set

- Strongly consistent set:
  - no message in transit during interval spanned by checkpoints
- Consistent set
  - each message recorded as received is also recorded as sent
  - i.e. lost messages are acceptable, orphans are not
  - $\{x_3, y_3, z_3\}$  is a consistent set
- A consistent set avoids the occurrence of domino effect because won't allow orphans

# Consistent set of checkpoint: algorithms

- Simple checkpointing algorithm :
  - if a process takes a checkpoint after sending every message, the set of most recent checkpoints is consistent
  - (some assumptions required, like atomicity of sending/receiving and checkpointing)
  - expensive
- Synchronous checkpointing algorithm:
  - Basic idea: all processes coordinate their actions in taking local checkpoints
  - Assumptions: FIFO channels, end-to-end protocol to deal with loss of messages (example: sliding window protocol)
  - Similar algorithm exists for synchronous rollback that avoids livelock
- Asynchronous algorithms:
  - Basic idea: no coordination at all, at rollback we'll figure it out ...

# Synchronous checkpointing algorithm

- Assumptions: FIFO, reliable communication
- Algorithm works in two phases:
  - Phase I: process  $P_i$  takes a tentative checkpoint and asks all others to do the same. If everybody else is successful, then  $P_i$  decides all tentative checkpoints should be made permanent
  - Phase II:  $P_i$  informs others of its decision (discard or commit)
- The set of checkpoints is consistent because:
  - either none or all processes take part
  - for the set to be inconsistent if a message is recorded received but not sent. But no process will send messages after being asked to take a tentative checkpoint until notification from  $P_i$

# Synchronous recovery

- Analogous to synchronous checkpointing:
  - Phase I: process  $P_i$  starts a vote for everybody to start a rollback. A process may vote “no” if already busy with a checkpoint or another rollback. If vote is a unanimous “yes”, then  $P_i$  decides all processes should rollback
  - Phase II:  $P_i$  informs others of its decision. Upon receiving its notification, processes start rollback
- Correctness:
  - all processes take same action - either rollback or continue
  - if all restart, then they resume execution in a consistent state (thanks to the synchronous checkpointing algorithm)

# Asynchronous algorithms

- Asynchronous approach:
  - don't synchronize actions - local checkpoints taken independently by each site
  - in the event of a rollback, search most recent consistent set between available checkpoints
- Different trade offs:
  - Synchronous checkpointing
    - simplifies recovery because checkpoints are consistent
    - trade off: increased burden on regular operations
  - Asynchronous checkpointing:
    - simplifies checkpointing because processes work independently
    - trade off: recovery becomes more complicated

# Fault Tolerance

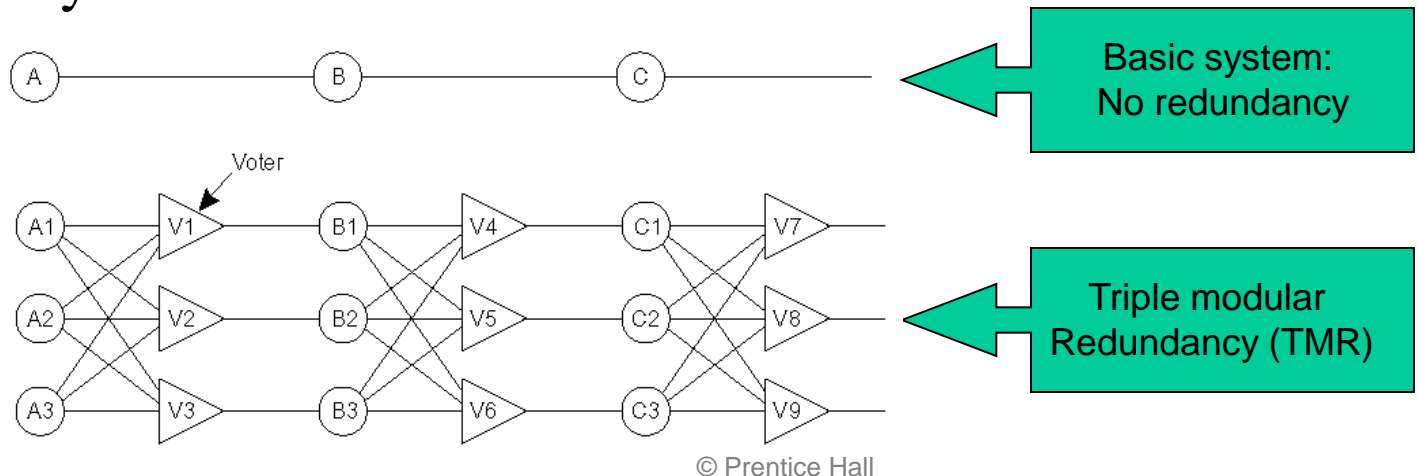
- To avoid disruption due to failure and to improve availability, systems are designed to be *fault-tolerant*
- Two broad categories of fault-tolerant systems are:
  - systems that *mask* failure
    - it continues to perform its function in the event of a failure
    - example: a system with redundant components
  - systems with a *well defined failure behavior*
    - may or may not perform its function; however it facilitate recovery
    - example: no changes made to a databases by a transaction until transaction successfully commits

# Fault-tolerance in o.s and distributed systems design

- Techniques employed in the design of fault-tolerant distributed systems:
  - voting protocols
    - Objective: masking of failures
  - commit protocols
    - Objective: well defined behavior in case of failure
  - Agreement protocols
    - Objective: reaching an agreement in presence of faulty information
- We are going to study one example of each one of these approaches

# Fault masking by Redundancy

- Redundancy is a key technique for masking faults:
  - Information redundancy (extra bits of information)
  - Time redundancy (same action performed several times)
  - Physical redundancy (extra equipment or processes; see example below)
- Voting mechanisms are used to mask faults using information redundancy



# Voting protocols

- Redundancy in the form of data replication is often used in distributed systems:
  - replicated DNS data at different sites
  - distributed file systems with multiple copies per file
- We will study a distributed voting mechanism
- Note distinction between voting vs. majority
  - majority is simple case of voting
  - voting is generally more flexible - allows differentiated read/write quorums, version management

# Gifford's voting algorithm

- Scenario: suppose you want to solve the readers/writers problem in an unreliable distributed system
  - data replicated at several sites to provide fault tolerance
  - each site is assigned one or more votes
  - each replica maintains its own version number
  - local Lock Managers implement multiple readers/single writer policy
- Reading (writing) permitted only if a minimum number of votes are collected
  - requesting process verifies reaching of *read/write quorum*
  - Timeouts employed in vote collection to prevent undefined waits in case of site failure

# The algorithm

- Site  $i$  gets lock from local Lock Manager then sends `Vote_Request` to all other sites
- site  $j$  checks with local Lock Manager, then replies sending version number  $VN_j$  and number of votes  $V_j$  to site  $i$
- site  $i$  adds votes and verifies that total is higher than read quorum  $r$  (write quorum  $w$ )
- if quorum reached, site  $i$  updates local copy of data if not current then proceeds to read/write it
  - Request aborted and locks released if quorum not reached
- after write is completed, site  $i$  updates  $VN_i$  and then sends the updated data and  $VN_i$  to all sites that replied