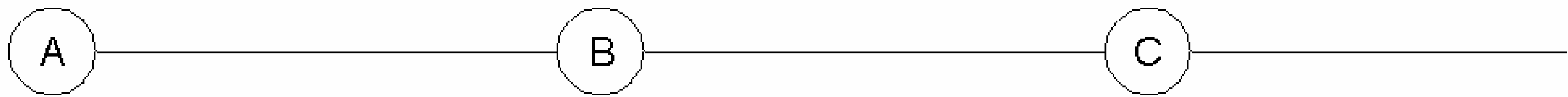
More on distributed systems

- In the past few weeks we have seen how to solve some interesting problems that arise in distributed systems
  - Process synchronization, logical clocks, mutual exclusion
- We are going to focus on other problems of practical relevance
  - For example, what about fault tolerance? Or coordination (i.e. reaching a common decision)?

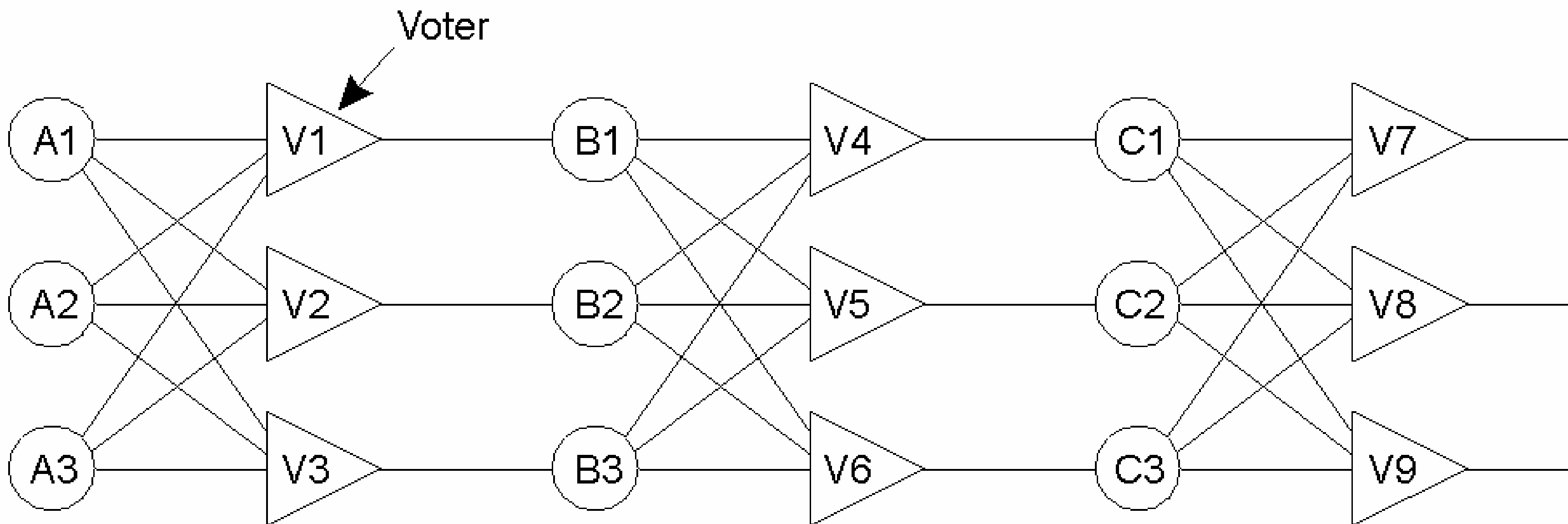
Fault Tolerance

- Fault tolerance is the ability of a system to continue operating despite faults
  - Fault tolerance is strongly related to the more general concept of dependability
- Dependability related concepts:
  - Availability
  - Reliability
  - Safety
  - Maintainability
- We will focus on software approaches to fault tolerance

Failure Masking by Redundancy



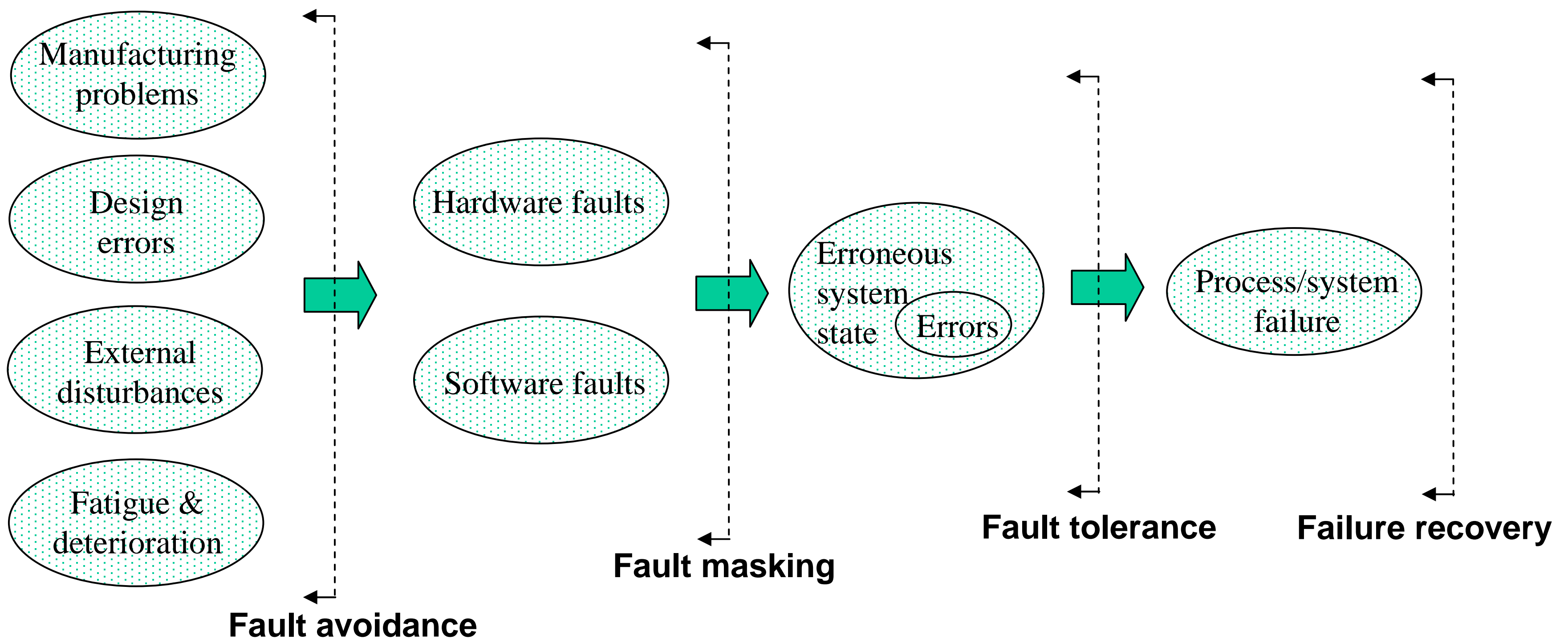
(a)



(b)

Cause-effect relationship

- Approaches to dealing with faults:
  - fault avoidance, fault masking, fault tolerance

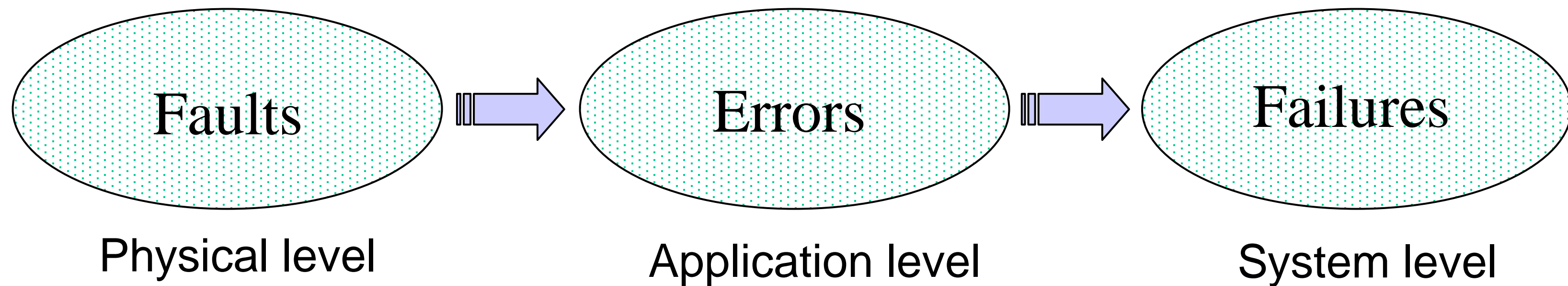


Recovery: definition

- Recovery refers to restoring a system to its normal operational state
- Simple case: restarting a failed computer, or a failed process
  - issues: recovering resources allocated to failed process, undoing uncomplete data modification, restarting execution from point of failure
- Distributed systems achieve increased availability through replication
  - Issues: recovering from inconsistency of data replicas in case of failure, fault/inconsistency isolation

Recovery: basic concepts

- Fault, error, failure:



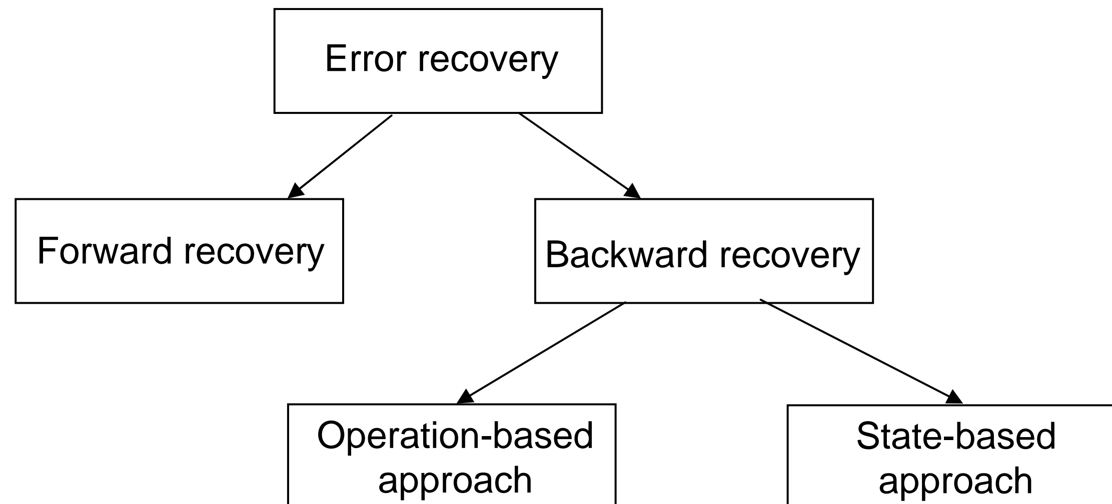
- Examples:
  - Physical level: logical gate stuck at “0”, input out of range
  - Application level: wrong value in program, seg fault
  - System level: application aborted, machine crash

Classification of failure

- Depending on what breaks, we can have:
  - Process failure
    - deadlocks, time-outs, wrong input, seg faults, ...
  - System failure
    - power failure, CPU failure, memory failure, bus failure, ..
  - Secondary storage failure
    - head crash, dust, parity error, ...
  - Communication failure
    - switching node failure, link breakage/excess noise, ..

Failure recovery

- Error: manifestation of a fault in the system
- Failure recovery: restoring an erroneous state to an error-free status



Forward recovery

- Assumption: errors can be detected and completely characterized
  - in this case the process (or system) can take corrective actions and then move forward
  - example: exception handling
- Tradeoffs:
  - lower overhead than backward recovery
  - can deal with unrecoverable components of the state
  - however not as general as backward recovery

Backward recovery

- Assumption: previous state can be restored
  - a process is restored to a previous state (*recovery point*)
  - information needed for recovery is stored on *stable* storage (i.e. failure-surviving storage)
- Operation-based approach
  - a *log* (a.k.a. *audit trail*) records state changes and allows reconstruction of the desired state
- State-based approach
  - a snapshot of the state is taken periodically

Operation-based approach

- "*Updating-in-place*":
  - every update (write) operation on an object also writes enough information on log to undo/redo the operation
    - DO: update object + writes old state and new state in log record
    - UNDO: undoes DO operation by using old state in log record
    - REDO: redoes DO operation using new state information in log
- "*Write-ahead-log*"
  - same as above, but with changes to make it more crash-safe:
    - updates object only after UNDO log is recorded
    - REDO and UNDO logs are recorded before committing the updates

State-based approach

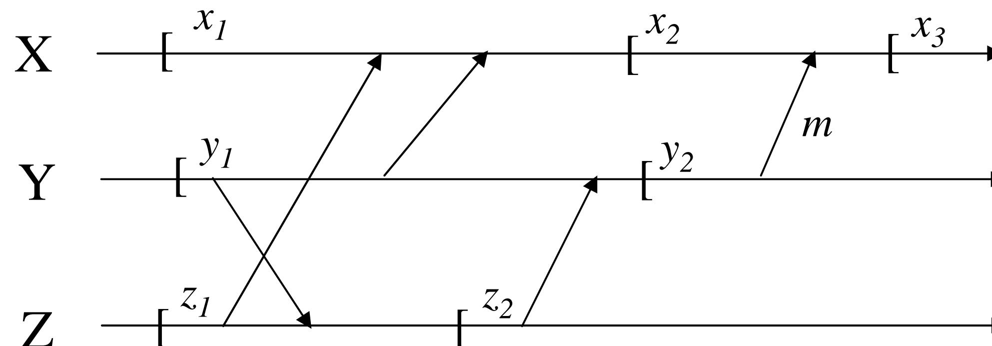
- Previous state snapshot is used for recovery
  - the snapshot is often referred to as a *checkpoint*
  - restoring the prior state is often referred to as *rolling back* the process
  - frequency: tradeoff between overhead of checkpointing and overhead of repeating execution of process/system
- Shadow Pages: partial state saved
  - only page where object being modified is saved ("shadowed") on stable storage
  - upon failure, the shadow copy is restored

Recovery in distributed systems

- Recovery is harder in distributed systems due to the presence of messages in transit
- Bad things can happen if checkpointing does not take into account messages:
  - orphan messages and domino effect
  - Lost messages
  - potential for livelocks
- Remedies:
  - (strongly) consistent set of checkpoints

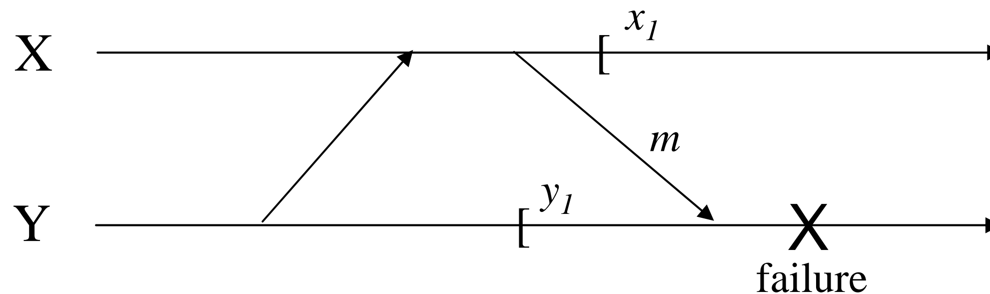
Orphan messages

- Orphan message: suppose Y fails after sending *m* and rolls back to y_2
  - receiving of *m* is recorded (in x_3) but the sending is not
  - X must be rolled back to undo the effects of the X-Y interaction
- Domino effect: suppose Z rolls back ...



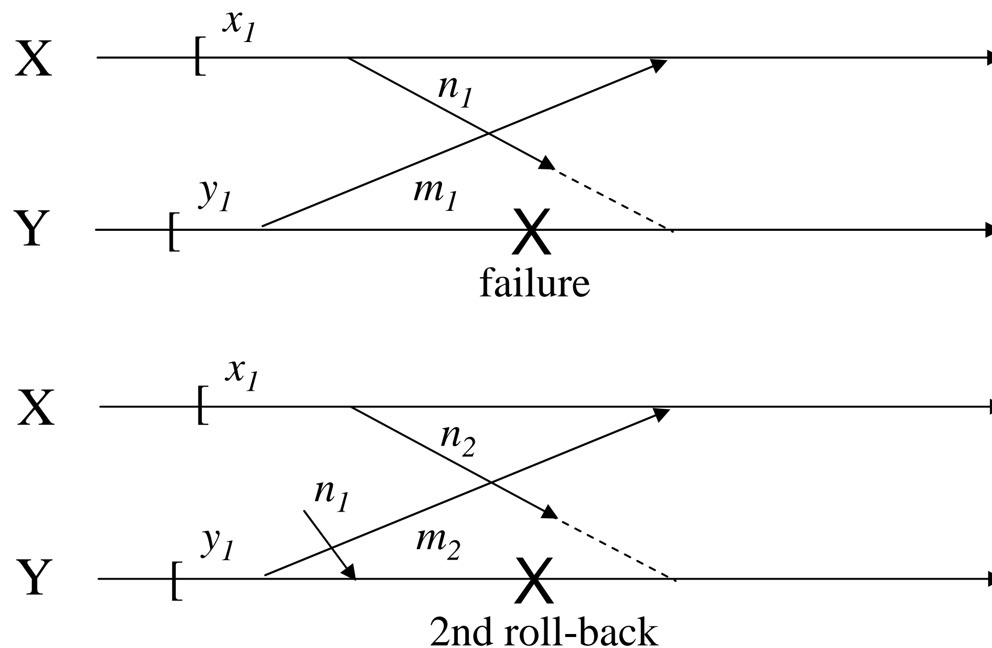
Lost messages

- Lost message: suppose Y rolls back after receiving *m*
  - the sending of *m* is recorded (in x_1) but not its reception



The problem of livelocks

- Single failure → infinite number of rollbacks → livelock

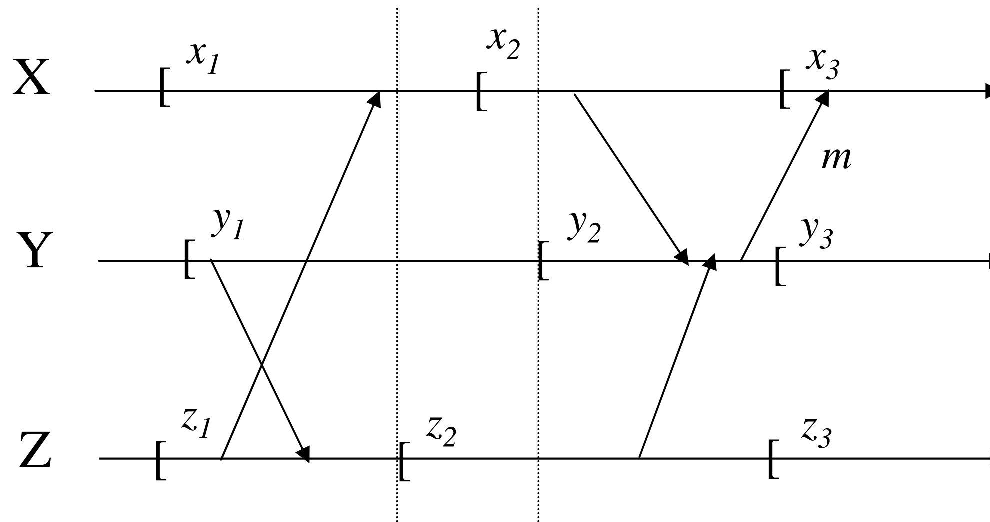


Global checkpoint

- A global checkpoint is a collection of local checkpoints
- How shown in the previous slides, taking local checkpoints at random can lead to problems
- Criteria to take safe global checkpoints
  - *(strongly) consistent* set of checkpoints

Consistent set of checkpoints

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



Strongly/simply 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

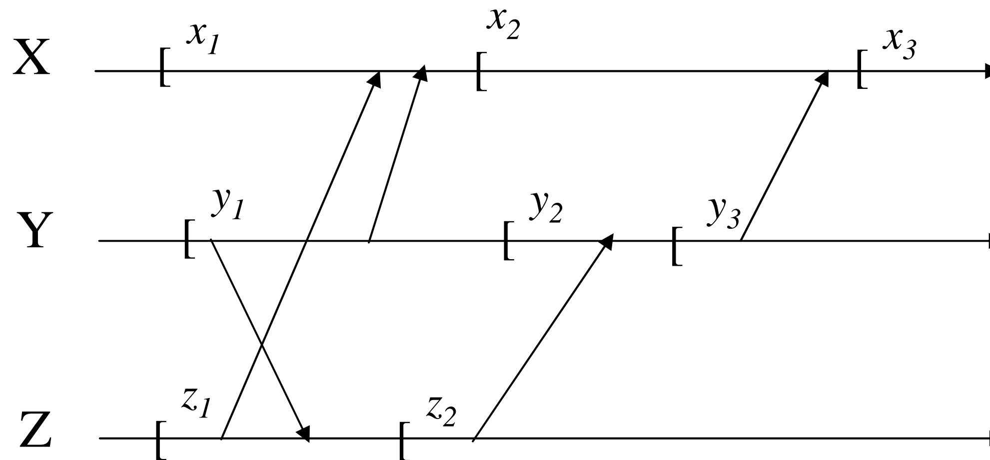
Synchronous recovery

- 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

Asynchronous algorithm example



- The latest set of checkpoints $\{x_3, y_3, z_2\}$ is not consistent.
  - The most recent set of consistent checkpoints (MRSCP) is $\{x_2, y_2, z_2\}$
- One way to find the MRSCP is the following:
  - Each checkpoint includes the tally of how many messages the processor has sent and received so far to/from everybody else;
  - Upon roll back, orphan messages can be detected by comparing the number of messages sent and received;
    - example: if *Y* tries to roll back to y_3 it will have one message sent to *X* on record – but *X* has two messages from *Y* on its record.
  - Keep trying new sets of checkpoints until one is found which is orphan-free.