Today
- Recovery
- CAP Theorem

Recovery
- Operations to be performed to move from an erroneous state to an error-free state
- Backward recovery: Go back to a previous correct state
  - E.g.: packet retransmission
- Forward recovery: Go to a new correct state
  - E.g.: Error-correction codes

Recovery techniques
- Checkpointing
- Message logging
- Rebooting
Checkpointing

- Periodically store state on stable storage
  - Mirrored/RAID disks, etc.
- At error-recovery, go back to the last checkpointed state
- Problem: How do we rollback so that all processes go back to a consistent global state?

Cuts in Global State Space

- Cut: Partition of events representing a global state
  - Set of last recorded event for each process

\[ \text{(a)} \quad \text{(b)} \]

Distributed Snapshot

- Consistent cut:
  - Receipt of a message \( m \) in the cut \( \implies \) sending of \( m \) also in the cut
  - If event \( a \) is in the cut, then all \( b \) s.t. \( b \succ a \) are in the cut
- Distributed Snapshot:
  - A consistent global state of the distributed system
- Recovery line: The most recent distributed snapshot

Independent Checkpointing

- Each process periodically checkpoints independently of other processes
- Upon a failure, work backwards to locate a consistent cut
- Domino effect: Cascading rollbacks
Coordinated Checkpointing

- Processes synchronize before checkpointing locally
- Synchronization techniques:
  - Two-phase protocol
  - Incremental snapshot

Two-phase protocol

- One process sends Checkpoint request
- Each recipient checkpoints current state, queues up new local messages
- Send checkpoint-done message

Incremental snapshot

- Checkpointing between causally related processes since last checkpointing
- Identify causally related processes incrementally
- Apply two-phase commit between these processes

Message Logging

- Checkpointing is expensive
  - Coordination, writing to stable storage
  - Too few checkpoints => can lose lot of state, need lot of recomputation, message passing
- Message logging
  - Take infrequent checkpoints
  - Log messages between checkpoints
- Recovery: Replay messages since last checkpoint
**Piecewise Deterministic Model**

- Execution of each process takes place in a series of intervals
  - Within each interval, the execution is deterministic
  - E.g.: sequence of instructions, message sending
- Start of each interval is a non-deterministic event
  - E.g.: receipt of a message
- Can replay the intervals if we log the non-deterministic events

**Orphan Process**

- Process whose state becomes inconsistent because of another process’s crash/recovery
  - Dependent on messages unlogged at crashed process
- Goal: Prevent orphan processes
  - When to log messages?

**Orphan Process Definition**

- Stable message: A message that cannot be lost
- DEP(m): Processes dependent on message m
  - Receivers of m, causally dependent on m
- COPY(m): Processes with non-stable copy of m
- Orphan process: P in DEP(m), no process in COPY(m)

**Message-Logging Schemes**

- Avoid orphan process:
  - no process in COPY(m) => no process in DEP(m)
- Pessimistic logging:
  - Ensures above property at time of message sending
  - At most one dependent process for any non-stable message m
- Optimistic logging:
  - Ensures above property after crash
  - Roll back all orphan processes to a state where they are not in DEP(m)
Rebooting

- Localize fault and reboot faulty component
  - Applied to software components
  - Requirement: Modularity, decoupling
- Basic idea: Fault dependent on a rare, transient event
- Recursive rebooting
  - Try shutting down the smallest faulty component first
  - Continue rebooting successively larger components

CAP Theorem

- C: Consistency
- A: Availability
- P: Tolerance to network partitions
- "2 of 3" rule: Can have only 2 of these properties

CAP Theorem - Revisited

- Is "2 of 3" rule misleading?

- What is meant by:
  - Consistency
  - Availability
  - Partitioning

CAP Choices - Factors

- Partitioning duration, frequency
  - E.g.: can have all 3 while network is connected
- Type of data, operations, users
  - E.g.: Yahoo PNUTS provides stronger A via local single-user master data
  - E.g.: Facebook prefers stronger C by maintaining a central master copy
**Partition Management**
- How to handle partitions when they occur?
- Key questions:
  - How to detect partitions?
  - How to operate during a partition?
  - How to recover after re-connection?

**Partition Detection**
- How would a node typically detect partitions?
- Related to communication latency
  - Network latency
  - Can be defined based on app latency requirements
- No global definition of partitioning
  - Different nodes may (or may not) detect network partition

**Partition Mode**
- How should the two sides operate under a partition?
  - Allow some operations. E.g.: those that do not conflict or could be resolved easily
  - Delay some and prohibit some. E.g.: those that need to be globally consistent
  - Maintain operation history. E.g.: version vectors

**Partition Recovery**
- Make state consistent on both sides
- Roll forward from a state before the partition
  - Use the logs to apply/merge operations
- How to merge conflicts?
  - Use version vectors
  - Might need manual intervention
  - Automated if we allow only limited operations. E.g.: only commutative operations
- Compensate for mistakes
  - Cancel a duplicate operation. E.g.: Refund user if charged twice