Lecture 8: Fault Tolerance

- Failure
- Reliable Communication
- Process Resilience
- Recovery

**Dependability**

**Availability:** system is ready to be used immediately

**Reliability:** system can run continuously without failure

**Safety:** when a system (temporarily) fails to operate correctly, nothing catastrophic happens

**Maintainability:** how easily a failed system can be repaired

Building a dependable system comes down to controlling failure and faults.

**Case Study: AWS Failure 2011**

- April 21, 2011
- EBS (Elastic Block Store) in US East region unavailable for about 2 days
- 13% of volumes in one availability zone got stuck
- led to control API errors and outage in whole region
- led to problems with EC2 instances and RDS in most popular region
- due to reconfig error and re-mirroring storm.

**AWS EBS Overview:**

- Region → Availability Zones
- Clusters → Nodes → Volumes
- Volume: replicated in cluster
- Control Plane Services: API for volumes for whole region
- Networks: primary, secondary

**What happened?**

- US east AZ
- network config problem
- re-mirroring storm
- CP API thread starvation
- node race condition
- CP election overload
Solution:
- Disconnect bad cluster
- Throttle re-mirroring
- Add more disk space
- Slowly un-throttle re-mirroring
- Volumes unstuck → reconnect cluster
- 0.07% data lost

Lessons learned:
- Back off
- Re-establish connectivity to previous replicas
- Shorter timeouts
- Snapshot stuck volumes
- CP: one AZ shouldn’t crash another AZ
- Make it easier to use multiple AZs

Failure

Terminology:
Failure: a system fails when it does not meet its promises or cannot provide its services in the specified manner
Error: part of the system state that leads to failure (i.e., it differs from its intended value)
Fault: the cause of an error (results from design errors, manufacturing faults, deterioration, or external disturbance)

Recursive:
- Failure can be a fault
- Manufacturing fault leads to disk failure
- Disk failure is a fault that leads to database failure
- Database failure is a fault that leads to email service failure

Total vs Partial Failure

Total Failure:
- All components in a system fail
- Typical in nondistributed system

Partial Failure:
- One or more (but not all) components in a distributed system fail
- Some components affected
- Other components completely unaffected
- Considered as fault for the whole system

Categorising Faults and Failures

Types of Faults:
Transient Fault: occurs once then disappear
Intermittent Fault: occurs, vanishes, reoccurs, vanishes, etc.
Permanent Fault: persists until faulty component is replaced

Types of Failures:
Process Failure: process proceeds incorrectly or not at all
Storage Failure: secondary storage is inaccessible
Communication Failure: communication link or node failure
**Failure Models**

- **Crash Failure**: a server halts, but works correctly until it halts
  - **Fail-Stop**: server will stop in a way that clients can tell that it has halted.
  - **Fail-Resume**: server will stop, then resume execution at a later time.
  - **Fail-Silent**: clients do not know server has halted
- **Omission Failure**: a server fails to respond to incoming requests
  - **Receive Omission**: fails to receive incoming messages
  - **Send Omission**: fails to send messages

- **Response Failure**: a server’s response is incorrect
  - **Value Failure**: the value of the response is wrong
  - **State Transition Failure**: the server deviates from the correct flow of control

- **Timing Failure**: a server’s response lies outside the specified time interval
- **Arbitrary Failure**: a server may produce arbitrary response at arbitrary times (aka Byzantine failure)

**Detecting Failure**

- **Failure Detector**: service that detects process failures
  - Answers queries about status of a process
- **Reliable**:
  - **Failed** – crashed
  - **Unsuspected** – hint
- **Unreliable**:
  - **Suspected** – may still be alive
  - **Unsuspected** – hint

**Synchronous systems**:
- **Timeout**
- Failure detector sends probes to detect crash failures

**Asynchronous systems**:
- **Timeout gives no guarantees**
- Failure detector can track suspected failures
- Combine results from multiple detectors
- **How to distinguish communication failure from process failure?**
  - Ignore messages from suspected processes
  - **Turn an asynchronous system into a synchronous one**
Fault Tolerance:

- System can provide its services even in the presence of faults

Goal:

- Automatically recover from partial failure
- Without seriously affecting overall performance

Techniques:

- Prevention: prevent or reduce occurrence of faults
- Masking: hide the occurrence of the fault
- Prediction: predict the faults that can occur and deal with them
- Recovery: restore an erroneous state to an error-free state

Failure Prevention

- Make sure faults don’t happen:
  - Quality hardware
  - Hardened hardware
  - Quality software

Failure Prediction

- Deal with expected faults:
  - Test for error conditions
  - Error handling code
  - Error correcting codes
    - checksums
    - erasure codes

Failure Masking

- Try to hide occurrence of failures from other processes

- Mask:
  1. Communication Failure → Reliable Communication
  2. Process Failure → Process Resilience
Redundancy:
- Information redundancy
- Time redundancy
- Physical redundancy

Two Army Problem:
Non-faulty processes but lossy communication.

Consensus with lossy communication is impossible.

Why does TCP work?

Reliable Communication
- Communication channel experiences failure
- Focus on masking crash (lost/broken connections) and omission (lost messages) failures

Reliable Point-to-Point Communication
- Reliable transport protocol (e.g., TCP)
  - Masks omission failure
  - Not crash failure
Example: Failure and RPC:
Possible failures:
- Client cannot locate server
- Request message to server is lost
- Server crashes after receiving a request
- Reply message from server is lost
- Client crashes after sending a request

How to deal with the various kinds of failure?

**Scalability of Reliable Multicast**

**Feedback Implosion:** sender is swamped with feedback messages

**Nonhierarchical Multicast:**
- Use NACKs
- Feedback suppression: NACKs multicast to everyone
- Prevents other receivers from sending NACKs if they’ve already seen one.
- Helps load on server
- Receivers have to be coordinated so they don’t all multicast NACKs at the same time
- Multicasting feedback also interrupts processes that successfully received message

**Hierarchical Multicast:**
Slide 25

**PROCESS RESILIENCE**

Protection against process failures

---

Slide 26

Groups:
- Organise identical processes into groups
  - Process groups are dynamic
  - Processes can be members of multiple groups
  - Mechanisms for managing groups and group membership
- Deal with all processes in a group as a single abstraction

Flat vs Hierarchical Groups:
- Flat group: all decisions made collectively
- Hierarchical group: coordinator makes decisions

---

Slide 27

**REPLICATION**

Create groups using replication

**Primary-Based:**
- Primary-backup
- Hierarchical group
- If primary crashes others elect a new primary

**Replicated-Write:**
- Active replication or Quorum
- Flat group
- Ordering of requests (atomic multicast problem)

**k-Fault Tolerance:**
- Can survive faults in k components and still meet its specifications
- k + 1 replicas enough if fail-silent (or fail-stop)
- 2k + 1 required if Byzantine

---

Slide 28

**STATE MACHINE REPLICATION**

---
Each replica executes as a state machine:
- state + input -> output + new state
- All replicas process same input in same order
- Deterministic: All correct replicas produce same output
- Output from incorrect replicas deviates

Input Messages:
- All replicas agree on content of input messages
- All replicas agree on order of input messages
- Consensus (also called Agreement)

What can cause non-determinism?

Atomic Multicast

A message is delivered to either all processes, or none

Requires agreement about group membership

Process Group:
- Group view: view of the group (list of processes) sender had when message sent
- Each message uniquely associated with a group
- All processes in group have the same view

View Synchrony:

A message sent by a crashing sender is either delivered to all remaining processes (crashed after sending) or to none (crashed before sending).

Reliable multicast by multiple point-to-point messages

- view changes and messages are delivered in total order Why?
Implementing View Synchrony:

**stable message**: a message that has been received by all members of the group it was sent to.

- Implemented using reliable point-to-point communication (TCP)
- Failure during multicast → only some messages delivered

**Slide 33**

Variants of the Agreement Problem

**Consensus**:
- Each process proposes a value
- Communicate with each other...
- All processes decide on same value
- For example, the maximum of all the proposed values

**Interactive Consistency**:
- All processes agree on a decision vector
- For example, the value that each of the processes proposed

**Byzantine Generals**:
- Commander proposes a value
- All other processes agree on the commander’s value

**Slide 35**

**Agreement**

Examples: Election, transaction commit/abort, dividing tasks among workers, mutual exclusion

- Previous algorithms assumed no faults
- What happens when processes can fail?
- What happens when communication can fail?
- What happens when byzantine failures are possible

We want all nonfaulty processes to reach and establish agreement (within a finite number of steps)

**Slide 34**

**Correctness of agreement**:

- **Termination**: All processes eventually decide
- **Agreement**: All processes decide on the same value

**Validity**:
- If the decided value was proposed by one of the processes
- IC: The decided value is a vector that reflects each of the processes proposed values
- BG: The decided value was proposed by the commander

**Slide 36**
**Consensus in a Synchronous System**

**Slide 37**
Assume:
- Execution in rounds
- Timeout to detect lost messages

**Byzantine Generals Problem:**
Reliable communication but faulty processes.
- $n$ generals (processes)
- $m$ are traitors (will send incorrect and contradictory info)
- Need to know everyone else's troop strength $g_i$
- Each process has a vector: $(g_1, \ldots, g_n)$
- (Note: this is actually interactive consistency)

**Slide 38**

**Slide 39**
Byzantine Generals Impossibility:
- If $m$ faulty processes then $2m + 1$ nonfaulty processes required for correct functioning

**Slide 40**
Byzantine agreement with Signatures:
- Digitally sign messages
- Cannot lie about what someone else said
- Avoids the impossibility result
- Can have agreement with 3 processes and 1 faulty
**Consensus in an Asynchronous System**

**Assume:**
- Arbitrary execution time (no rounds)
- Arbitrary message delays (can’t rely on timeout)

**Impossibility of Consensus with One Failure**

Impossible to guarantee consensus with $\geq 1$ faulty process

**Proof Outline:**
- Fischer, Lynch, Patterson (FLP) 1985
- The basic idea is to show circumstances under which the protocol remains forever indecisive
- Bivalent (any result is possible) vs univalent (only single result is possible) states
  1. There is always a bivalent start state
  2. Always possible to reach a bivalent state by delaying messages → no termination

In practice we can get close enough

---

**Consensus in Practice**

**Two Phase Commit:**
- Original assumption: No failure

**Failures can be due to:**
- Failure of communication channels:
  - Use timeouts
- Server failures:
  - Potentially blocking

**Two-phase commit with timeouts: Worker:**

- On timeout sends GetDecision.
Two-phase commit with timeouts: Coordinator:

- On timeout re-sends CanCommit, On GetDecision repeats decision.

Coordinator failure:
- When coordinator crashes start a new recovery coordinator
- Learn state of protocol from workers (what did they vote, what did they learn from coordinator)
- Finish protocol

Coordinator and Worker failure: Blocking 2PC:
- Recovery coordinator can’t distinguish between
  - All workers vote Commit and failed worker already committed
  - Failed worker voted Abort and rest of workers voted Commit
- So can’t make a decision

Three Phase Commit
- Vote: as in 2PC
- Pre-commit: coordinator sends vote result to all workers, workers acknowledge
- Commit: coordinator tells workers to perform vote action

Why does this work?

Raft
- Reliable, Replicated, Redundant, And Fault-Tolerant Consensus
- Goal: each node agrees on the same series of operations (log)
- Log ordered list of operations
- Leader node responsible for deciding how to add operations to the log
- Followers nodes that replicate the leader’s log
- Two subproblems
- Leader Election how to agree on who the leader is
- Log Replication how to replicate the leader’s log to the followers
**LEADER ELECTION**

**Term** the time during which a node is leader

**Candidate** node who wants to become leader

**Failed Leader:**
- Leader sends regular heartbeat to followers
- Follower sees no communication from leader (election timeout)
- Leader sees heartbeat from a later term (steps down as leader)

---

**Candidate:**
- Detects that leader has failed
- Sends Request Vote to all other nodes
- Nodes reply
  - Yes hasn’t voted yet this term
  - No has already voted this term
- Majority votes -> candidate becomes leader
- or Timeout -> new election
Log Replication

Slide 53

Log Replication

Slide 54

Log Replication

Slide 55

Log Replication

Slide 56

1. Client sends operation to leader
2. Leader appends to its log
3. Leader sends Append Entries message to followers
4. Followers acknowledge
5. Leader commits operation to log
**Paxos**

Goal: a collection of processes chooses a single proposed value in the presence of failure

**Proposer** proposes value to choose (leader)

**Acceptor** accept or reject proposed values

**Learner** any process interested in the result (chosen value) of the consensus

**Chosen Value:** value accepted by majority of acceptors

**Properties:**
- Only proposed values can be learned
- At most one value can be learned
- If a value has been proposed then eventually a value will be learned

**Using Paxos**

Use Paxos for:
- **Leader election:** choose a leader id
  - single paxos instance, elections starter(s) propose leader id, result in an agreed upon leader.
- **View synchrony:** order view changes
  - one paxos instance per view change, result in a view change order sequence number
- **Total order multicast:** order messages
  - one paxos instance per message, result in a message sequence number
- **State machine replication:** order operations
  - one paxos instance per operation, result in an operation sequence number

**Paxos Algorithm: 3 Phases**

Assuming no failures

**Phase 1: Propose:**
1. Propose: send a proposal \(<seq, value>\) to \(\geq N/2\) acceptors
2. Promise: acceptors reply.
   - accept (include last accepted value), promised = seq.

**Phase 2: Accept:**
1. Accept: when \(\geq N/2\) accept replies, proposer sends value (as received from acceptor or arbitrary):
2. Accepted: acceptors reply
   - accepted. Remember accepted value.

**Phase 3: Learn:**
1. Propagate value to Learners when \(\geq N/2\) accepted replies received.

**Simple Case**
**Failures**

What can go wrong before agreement is reached?

**Failure Model:**
- channel: lose, reorder, duplicate message
- process: crash (fail-stop, fail-resume)

**Slide 61**

**Failure Cases:**
- ➀ Acceptor fails
- ➁ Acceptor recovers/restarts
- ➂ Proposer fails
- ➃ Multiple proposers
  - New proposer
  - Proposer recovers/restarts

**Paxos Algorithm: 3 Phases**

**With Failures!**

**Phase 1: Propose:**
- ➀ Propose: send a proposal \(<seq, value>\) to \(\geq N/2\) acceptors
- ➁ Promise: acceptors reply.
  - reject if \(seq < seq\) of previously accepted value
  - else accept (include last accepted value) \(promised = seq\).

**Slide 62**

**Phase 2: Accept:**
- ➀ Accept: when \(\geq N/2\) accept replies, proposer sends value (as received from acceptor or arbitrary):
- ➁ Accepted: acceptors reply
  - reject if \(seq < promised\).
  - else accepted: Remember accepted value.

**Phase 3: Learn:**
- ➀ Propagate value to Learners when \(\geq N/2\) accepted received.

**Acceptor Fails**

- As long as a quorum still available
- → Restart: Must remember last accepted value(s)

**Slide 63**

**Proposer Fails**

- Elect a new leader
- Continue execution
- New proposer will choose any previously accepted value

**Slide 64**
MULTIPLE PROPOSERS

- For example: crashed proposer returns and continues
- Dueling proposers
- No guaranteed termination
- Heuristics to recognise situation and back off

OPTIMISATION AND MORE INFORMATION

Opportunities for optimisation:
- Reduce rounds
  - Phase 1: reject, return highest accepted seq
  - Phase 2: reject, return promised seq
- Reduce messages
  - Piggyback multiple requests and replies
  - Pre-propose multiple instances (assumes Proposer rarely fails)

More information:
Paxos Made Live - An Engineering Perspective  Experiences implementing Paxos for Google’s Chubby lock server. It turns out to be quite complicated.

FAILURE RECOVERY

Restoring an erroneous state to an error free state

Issues:
- Reclamation of resources:
  - Locks, buffers held on other nodes
- Consistency:
  - Undo partially completed operations prior to restart
- Efficiency:
  - Avoid restarting whole system from start of computation
Forward vs. Backward Error Recovery

Forward Recovery:
- Correct erroneous state without moving back to a previous state.
- Example: erasure correction - missing packet reconstructed from successfully delivered packets.
- Possible errors must be known in advance

Backward Recovery:
- Correct erroneous state by moving to a previously correct state.
- Example: packet retransmission when packet is lost.
- General purpose technique.
- High overhead.
- Error can reoccur.
- Sometimes impossible to roll back (e.g. ATM has already delivered the money).

General Approach:
- Restore process to recovery point.
- Restore system by restoring all active processes.

Specific Approaches:

- Operation-based recovery:
  - Keep log (or audit trail) of operations (like transactions).
  - Restore to recovery point by reversing changes.

- State-based recovery:
  - Store complete state at recovery point (checkpointing).
  - Restore process state from checkpoint (rolling back).

Log or checkpoint recorded on stable storage.

State-Based Recovery - Checkpointing:
- Take frequent checkpoints during execution.

Checkpointing:
- Pessimistic vs Optimistic:
  - Pessimistic: assumes failure, optimised toward recovery.
  - Optimistic: assumes infrequent failure, minimises checkpointing overhead.

- Independent vs Coordinated:
  - Coordinated: processes synchronise to create global checkpoint.
  - Independent: each process takes local checkpoints independently of others.

- Synchronous vs Asynchronous:
  - Synchronous: distributed computation blocked while checkpoint taken.
  - Asynchronous: distributed computation continues while checkpoint taken.

Checkpointing Overhead:
- Frequent checkpointing increases overhead.
- Infrequent checkpointing increases recovery cost.

Decreasing Checkpointing Overhead:

- Incremental checkpointing: Only write changes since last checkpoint.
  - Write-protect whole address space.
  - On write-fault mark page as dirty and unprotect.
  - On checkpoint only write dirty pages.

- Asynchronous checkpointing: Use copy-on-write to checkpoint while execution continues.
  - Easy with UNIX fork().

- Compress checkpoints: Reduces storage and I/O cost at the expense of CPU time.
**RECOVERY IN DISTRIBUTED SYSTEMS**

> Failed process may have causally affected other processes
> Upon recovery of failed process, must undo effects on other processes
> Must roll back all affected processes
> All processes must establish recovery points
> Must roll back to a consistent global state

**Domino Effect:**

- \( P_1 \) fails \( \rightarrow \) roll back: \( P_1 \bowtie R_{13} \)
- \( P_2 \) fails \( \rightarrow P_2 \bowtie R_{22} \)
  - Orphan message \( m \) is received but not sent \( \rightarrow P_1 \bowtie R_{12} \)
- \( P_3 \) fails \( \rightarrow P_3 \bowtie R_{13} \)
  - \( P_2 \bowtie R_{21} \)

Messages dependencies plus independent checkpointing may force system to roll back to initial state

**Message Loss:**

- Failure of \( P_2 \) \( \rightarrow P_2 \bowtie R_{21} \)
- Message \( m \) is now recorded as sent (by \( P_1 \)) but not received (by \( P_2 \)), and \( m \) will never be received after rollback
- Message \( m \) is lost
- Whether \( m \) is lost due to rollback or due to imperfect communication channels is indistinguishable!
- Require protocols resilient to message loss

**Livelock:**

- \( P_2 \downarrow \rightarrow P_2 \bowtie R_{21} \rightarrow P_1 \bowtie R_{11} \)
  - Note: \( n_1 \) in transit
- Pre-rollback message \( n_1 \) is received after rollback
- Forces another rollback \( P_2 \bowtie R_{21}, P_1 \bowtie R_{11} \), can repeat indefinitely
**Consistent Checkpointing**

Consistent Cut:

- Idea: collect local checkpoints in a coordinated way.
  - Set of local checkpoints forms a global checkpoint.
  - A global checkpoint represents a consistent system state.

- \( R_{11}, R_{21}, R_{31} \) form a strongly consistent checkpoint:
  - No information flow during checkpoint interval

- \( R_{12}, R_{22}, R_{32} \) form a consistent checkpoint:
  - All messages recorded as received must be recorded as sent

**Strongly consistent checkpointing** requires quiescent system
→ Potentially long delays during blocking checkpointing

**Consistent checkpointing** requires dealing with message loss
- Not a bad idea anyway, as otherwise each lost message would result in a global rollback
- Note that a consistent checkpoint may not represent an actual past system state

How to take a consistent checkpoint?:
- Simple solution: Each process checkpoints immediately after sending a message
  - High overhead
  - Reducing this to checkpointing after \( n \) messages, \( n > 1 \), is not guaranteed to produce a consistent checkpoint!
  - Require some coordination during checkpointing

**Synchronous Checkpointing**

Processes coordinate local checkpointing so that most recent local checkpoints constitute a consistent checkpoint

Assumptions:
- Communication is via FIFO channels.
- Message loss dealt with via
  - Protocols (such as sliding window), or
  - Logging of all sent messages to stable storage
- Network will not partition

Local checkpoints:
- **permanent**: part of a global checkpoint
- **tentative**: may or may not become permanent
SYNCHRONOUS ALGORITHM

- Global checkpoint initiated by a single coordinator
- Based on 2PC

First Phase:
1. Coordinator $P_i$ takes tentative checkpoint
2. $P_i$ sends message to all other processes $P_j$ to take tentative checkpoint
3. $P_j$ reply to $P_i$ whether succeeded in taking tentative checkpoint
4. $P_i$ receives true reply from each $P_j$ → decides to make permanent
   $P_i$ receives at least one false → decides to discard the tentative checkpoints

Second Phase:
1. Coordinator $P_i$ informs all other processes $P_j$ of decision
2. $P_j$ convert or discard tentative checkpoints accordingly

Consistency ensured because no messages sent between two checkpoint messages from $P_i$.

REDUNDANT CHECKPOINTS

Algorithm performs unnecessary checkpoints

First Phase:
1. Coordinator sends "r" messages to all other processes to ask them to rollback
2. Each process replies true, unless already in checkpoint or rollback
3. If all replies are true, coordinator decides to rollback, otherwise continue

Second Phase:
1. Coordinator sends decision to other processes
2. Processes receiving this message perform corresponding action

Slide 83

ROLLBACK RECOVERY

First Phase:
1. Coordinator sends "r" messages to all other processes to ask them to rollback
2. Each process replies true, unless already in checkpoint or rollback
3. If all replies are true, coordinator decides to rollback, otherwise continue

Second Phase:
1. Coordinator sends decision to other processes
2. Processes receiving this message perform corresponding action
**Homework**

- Look up a recent failure of a large distributed system. What went wrong? How could the problem have been avoided? What lessons were learned?
- Find a Paxos or Raft library and implement a replicated state machine using it.

**Hacker’s edition:**
- Implement the Paxos or Raft library (e.g., in Erlang).

---

**Reading List**

*Optional*

- *Paxos Made Live - An Engineering Perspective* Experiences implementing Paxos for Google’s Chubby lock server. It turns out to be quite complicated.