Today: Fault Tolerance

- Agreement in presence of faults
  - Two army problem
  - Byzantine generals problem
- Reliable communication
- Distributed commit
  - Two phase commit
  - Three phase commit
- Paxos
- Failure recovery
  - Checkpointing
  - Message logging

Fault Tolerance

- Single machine systems
  - Failures are all or nothing
    - OS crash, disk failures
- Distributed systems: multiple independent nodes
  - Partial failures are also possible (some nodes fail)
- Question: Can we automatically recover from partial failures?
  - Important issue since probability of failure grows with number of independent components (nodes) in the systems
  - \( \text{Prob}(\text{failure}) = \text{Prob}(\text{Any one component fails}) = 1 - \text{P(no failure)} \)
A Perspective

• Computing systems are not very reliable
  – OS crashes frequently (Windows), buggy software, unreliable hardware, software/hardware incompatibilities
  – Until recently: computer users were “tech savvy”
    • Could depend on users to reboot, troubleshoot problems
  – Growing popularity of Internet/World Wide Web
    • “Novice” users
    • Need to build more reliable/dependable systems
  – Example: what is your TV (or car) broke down every day?
    • Users don’t want to “restart” TV or fix it (by opening it up)

• Need to make computing systems more reliable
  – Important for online banking, e-commerce, online trading, webmail…

Basic Concepts

• Need to build dependable systems
• Requirements for dependable systems
  – Availability: system should be available for use at any given time
    • 99.999 % availability (five 9s) => very small down times
  – Reliability: system should run continuously without failure
  – Safety: temporary failures should not result in a catastrophic
    • Example: computing systems controlling an airplane, nuclear reactor
  – Maintainability: a failed system should be easy to repair
Basic Concepts (contd)

- Fault tolerance: system should provide services despite faults
  - Transient faults
  - Intermittent faults
  - Permanent faults

### Failure Models

<table>
<thead>
<tr>
<th>Type of failure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash failure</td>
<td>A server halts, but is working correctly until it halts</td>
</tr>
<tr>
<td>Omission failure</td>
<td>A server fails to respond to incoming requests</td>
</tr>
<tr>
<td>Receive omission</td>
<td>A server fails to receive incoming messages</td>
</tr>
<tr>
<td>Send omission</td>
<td>A server fails to send messages</td>
</tr>
<tr>
<td>Timing failure</td>
<td>A server's response lies outside the specified time interval</td>
</tr>
<tr>
<td>Response failure</td>
<td>The server's response is incorrect</td>
</tr>
<tr>
<td>Value failure</td>
<td>The value of the response is wrong</td>
</tr>
<tr>
<td>State transition fail</td>
<td>The server deviates from the correct flow of control</td>
</tr>
<tr>
<td>Arbitrary failure</td>
<td>A server may produce arbitrary responses at arbitrary times</td>
</tr>
</tbody>
</table>

- Different types of failures.
Failure Masking by Redundancy

- Triple modular redundancy.

Agreement in Faulty Systems

- How should processes agree on results of a computation?
- \textit{K-fault tolerant}: system can survive \( k \) faults and yet function
- Assume processes fail silently
  - Need \((k+1)\) redundancy to tolerant \( k \) faults
- \textit{Byzantine failures}: processes run even if sick
  - Produce erroneous, random or malicious replies
    - Byzantine failures are most difficult to deal with
  - Need ? Redundancy to handle Byzantine faults
Byzantine Faults

- Simplified scenario: two perfect processes with unreliable channel
  - Need to reach agreement on a 1 bit message
- Two army problem: Two armies waiting to attack
  - Each army coordinates with a messenger
  - Messenger can be captured by the hostile army
  - Can generals reach agreement?
  - Property: Two perfect process can never reach agreement in presence of unreliable channel
- Byzantine generals problem: Can N generals reach agreement with a perfect channel?
  - M generals out of N may be traitors

Byzantine Generals Problem

Recursive algorithm by Lamport
- The Byzantine generals problem for 3 loyal generals and 1 traitor.
  a) The generals announce their troop strengths (in units of 1 kilosoldiers).
  b) The vectors that each general assembles based on (a)
  c) The vectors that each general receives in step 3.
Byzantine Generals Problem Example

- The same as in previous slide, except now with 2 loyal generals and one traitor.
- Property: With \( m \) faulty processes, agreement is possible only if \( 2m+1 \) processes function correctly out of \( 3m+1 \) total processes. [Lamport 82]
  - Need more than two-thirds processes to function correctly

Byzantine Fault Tolerance

- Detecting a faulty process is easier
  - \( 2k+1 \) to detect \( k \) faults

- Reaching agreement is harder
  - Need \( 3k+1 \) processes (\( 2/3 \text{rd} \) majority needed to eliminate the faulty processes)

- Implications on real systems:
  - How many replicas?
  - Separating agreement from execution provides savings
Reaching Agreement

• If message delivery is unbounded,
  – No agreement can be reached even if one process fails
  – Slow process indistinguishable from a faulty one

• BAR Fault Tolerance
  – Until now: nodes are byzantine or collaborative
  – New model: Byzantine, Altruistic and Rational
  – Rational nodes: report timeouts etc

Reliable One-One Communication

• Issues were discussed in Lecture 3
  – Use reliable transport protocols (TCP) or handle at the application layer
• RPC semantics in the presence of failures
• Possibilities
  – Client unable to locate server
  – Lost request messages
  – Server crashes after receiving request
  – Lost reply messages
  – Client crashes after sending request
Reliable One-Many Communication

- Reliable multicast
  - Lost messages => need to retransmit
- Possibilities
  - ACK-based schemes
    - Sender can become bottleneck
  - NACK-based schemes

Atomic Multicast

- Atomic multicast: a guarantee that all process received the message or none at all
  - Replicated database example
  - Need to detect which updates have been missed by a faulty process
- Problem: how to handle process crashes?
- Solution: group view
  - Each message is uniquely associated with a group of processes
    - View of the process group when message was sent
    - All processes in the group should have the same view (and agree on it)
Implementing Virtual Synchrony in Isis

(a) Process 4 notices that process 7 has crashed, sends a view change
(b) Process 6 sends out all its unstable messages, followed by a flush message
(c) Process 6 installs the new view when it has received a flush message from everyone else

<table>
<thead>
<tr>
<th>Multicast</th>
<th>Basic Message Ordering</th>
<th>Total-Ordered Delivery?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliable multicast</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>FIFO multicast</td>
<td>FIFO-ordered delivery</td>
<td>No</td>
</tr>
<tr>
<td>Causal multicast</td>
<td>Causal-ordered delivery</td>
<td>No</td>
</tr>
<tr>
<td>Atomic multicast</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>FIFO atomic multicast</td>
<td>FIFO-ordered delivery</td>
<td>Yes</td>
</tr>
<tr>
<td>Causal atomic multicast</td>
<td>Causal-ordered delivery</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Distributed Commit

- Atomic multicast example of a more general problem
  - All processes in a group perform an operation or not at all
  - Examples:
    - Reliable multicast: Operation = delivery of a message
    - Distributed transaction: Operation = commit transaction

- Problem of distributed commit
  - All or nothing operations in a group of processes

- Possible approaches
  - Two phase commit (2PC) [Gray 1978]
  - Three phase commit

## Two Phase Commit

- Coordinator process coordinates the operation
- Involves two phases
  - Voting phase: processes vote on whether to commit
  - Decision phase: actually commit or abort

![Two Phase Commit Diagram](chart)
Implementing Two-Phase Commit

actions by coordinator:

while START_2PC to local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected {
    wait for any incoming vote;
    if timeout {
        while GLOBAL_ABORT to local log;
multicast GLOBAL_ABORT to all participants;
exit;
    }
    record vote;
}
if all participants sent VOTE_COMMIT and coordinator votes COMMIT{
write GLOBAL_COMMIT to local log;
multicast GLOBAL_COMMIT to all participants;
} else {
write GLOBAL_ABORT to local log;
multicast GLOBAL_ABORT to all participants;
}

• Outline of the steps taken by the coordinator in a two phase commit protocol

actions by participant:

write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
    write VOTE_ABORT to local log;
exit;
}
if participant votes COMMIT {
    write VOTE_COMMIT to local log;
send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
        multicast DECISION_REQUEST to other participants;
wait until DECISION is received; /* remain blocked */
write DECISION to local log;
    }
    if DECISION == GLOBAL_COMMIT
write GLOBAL_COMMIT to local log;
else if DECISION == GLOBAL_ABORT
write GLOBAL_ABORT to local log;
} else {
write VOTE_ABORT to local log;
send VOTE_ABORT to coordinator;
}

actions for handling decision requests: /*executed by separate thread */

while true {
wait until any incoming DECISION_REQUEST is received; /* remain blocked */
    read most recently recorded STATE from the local log;
    if STATE == GLOBAL_COMMIT
send GLOBAL_COMMIT to requesting participant;
else if STATE == INIT or STATE == GLOBAL_ABORT
send GLOBAL_ABORT to requesting participant;
else
skip; /* participant remains blocked */
Recovering from a Crash

- If INIT: abort locally and inform coordinator
- If Ready, contact another process Q and examine Q’s state

<table>
<thead>
<tr>
<th>State of Q</th>
<th>Action by P</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMIT</td>
<td>Make transition to COMMIT</td>
</tr>
<tr>
<td>ABORT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>INIT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>READY</td>
<td>Contact another participant</td>
</tr>
</tbody>
</table>

Three-Phase Commit

Two phase commit: problem if coordinator crashes (processes block)
Three phase commit: variant of 2PC that avoids blocking
Replication for Fault Tolerance

• Basic idea: use replicas for the server and data

• Technique 1: split incoming requests among replicas
  – If one replica fails, other replicas take over its load
  – Suitable for crash fault tolerance (each replica produces correct results when it is us).

• Technique 2: send each request to all replicas
  – Replicas vote on their results and take majority result
  – Suitable for BFT (a replica can produce wrong results)
    • 2PC, 3PC, Paxos are techniques

Consensus, Agreement

• Consensus protocols
• Achieve reliability in presence of faulty processes
  – requires processes to agree on data value needed for computation
  – Examples: whether to commit a transaction, agree on identity of a leader, atomic broadcasts, distributed locks
• Properties of a consensus protocol with fail-stop failures
  – Agreement: every correct process agrees on same value
  – Termination: every correct process decides some value
  – Validity: If all propose v, all correct processes decides v
  – Integrity: Every correct process decided at most one value and if it decides v, someone must have proposed v.
2PC, 3PC Problems

- Both have problems in presence of failures
  - Safety is ensured but liveness is not
- 2PC
  - must wait for all nodes and coordinator to be up
  - all nodes must vote
  - coordinator must be up
- 3PC
  - handles coordinator failure
  - but network partitions are still an issue
- Paxos: how to reach consensus in distributed systems that can tolerate non-malicious failures?

Paxos: fault-tolerant agreement

- Paxos lets nodes agree on same value despite:
  - node failures, network failures and delays
- Use cases:
  - Nodes agree X is primary (or leader)
  - Nodes agree Y is last operation (order operations)
- General approach
  - One (or more) nodes decides to be leader (aka proposer)
  - Leader proposes a value and solicits acceptance from others
  - Leader announces result or tries again
- Proposed independently by Lamport and Liskov
  - Widely used in real systems in major companies
Paxos Requirements

• Safety (Correctness)
  – All nodes agree on the same value
  – Agreed value X was proposed by some node

• Liveness (fault-tolerance)
  – If less than N/2 nodes fail, remaining nodes will eventually reach agreement
  – Liveness not guaranteed if steady stream of failures

• Why is agreement hard?
  – Network partitions
  – Leader crashes during solicitation or after deciding but before announcing results,
  – New leader proposes different value from already decided value,
  – More than one node becomes leader simultaneously....

Paxos Setup

• Entities: Proposer (leader), acceptor, learner
  – Leader proposes value, solicits acceptance from acceptors
  – Acceptors are nodes that want to agree; announce chosen value to learners

• Proposals are ordered by proposal #
  – node can choose any high number to try to get proposal accepted
  – An acceptor can accept multiple proposals
    • If prop with value v chosen, all higher proposals have value v

• Each node maintains
  – n_a, v_a: highest proposal # and accepted value
  – n_h : highest proposal # seen so far
  – my_n: my proposal # in current Paxox
Paxos operation: 3 phase protocol

- **Phase 1 (Prepare phase)**
  - A node decides to be a leader and propose
  - Leader chooses $my_n > n_h$
  - Leader sends $<\text{prepare, my}_n>$ to all nodes
  - Upon receiving $<\text{prepare, n}>$ at acceptor
    - If $n < n_h$
      - reply $<\text{prepare-reject}>$ /* already seen higher # proposal */
    - Else
      - $n_h = n$ /* will not accept prop lower than n */
      - reply $<\text{prepare-ok, n_a, v_a}>$ /* send back previous prop, value*/
      - /* can be null, if first */

- **Phase 2 (accept phase)**
  - If leader gets prepare-ok from majority
    - $V = \text{non-empty value from highest n_a received}$
    - If $V = \text{null}$, leader can pick any $V$
    - Send $<\text{accept, my}_n, V>$ to all nodes
  - If leader fails to get majority prepare-ok
    - delay and restart Paxos
  - Upon receiving $<\text{accept, n, V}>$
    - If $n < n_h$
      - reply with $<\text{accept-reject}>$
    - else
      - $n_a = n$; $v_a = V$; $n_h = h$; reply $<\text{accept-ok}>$
Paxos Operation

• **Phase 3 (decide)**
  – If leader gets accept-ok from majority
    • Send <decide, v_a> to all learners
  – If leader fails to get accept-ok from a majority
    • Delay and restart Paxos

• **Properties**
  – P1: any proposal number is unique
  – P2: any two set of acceptors have at least one node in common
  – P3: value sent in phase 2 is value of highest numbered proposal received in responses in phase 1

Paxos Example
Issues

• Network partitions:
  – With one partition, will have majority on one side and can come to agreement (if nobody fails)

• Timeouts
  – A node has max timeout for each message
  – Upon timeout, declare itself as leader and restart Paxos

• Two leaders
  – Either one leader is not able to decide (does not receive majority accept-oks since nodes see higher proposal from other leader) OR
  – one leader causes the other to use it value

• Leader failures: same as two leaders or timeout occurs

Recovery

• Techniques thus far allow failure handling
• Recovery: operations that must be performed after a failure to recover to a correct state
• Techniques:
  – Checkpointing:
    • Periodically checkpoint state
    • Upon a crash roll back to a previous checkpoint with a consistent state
Independent Checkpointing

- Each processes periodically checkpoints independently of other processes
- Upon a failure, work backwards to locate a consistent cut
- Problem: if most recent checkpoints form inconsistent cut, will need to keep rolling back until a consistent cut is found
- Cascading rollbacks can lead to a domino effect.

Coordinated Checkpointing

- Take a distributed snapshot [discussed in Lec 11]
- Upon a failure, roll back to the latest snapshot
  - All process restart from the latest snapshot
Logging

- Logging: a common approach to handle failures
  - Log requests / responses received by system on separate storage device / file (stable storage)
    - Used in databases, filesystems, ...
- Failure of a node
  - Some requests may be lost
  - Replay log to “roll forward” system state

Message Logging

- Checkpointing is expensive
  - All processes restart from previous consistent cut
  - Taking a snapshot is expensive
  - Infrequent snapshots => all computations after previous snapshot will need to be redone [wasteful]
- Combine checkpointing (expensive) with message logging (cheap)
  - Take infrequent checkpoints
  - Log all messages between checkpoints to local stable storage
  - To recover: simply replay messages from previous checkpoint
    - Avoids recomputations from previous checkpoint