Basics of Fault Tolerance

Failure of Systems

- Failure of a system – when the system deviates from its specifications
- Failure of a system may cause it to fail to provide the service it provides
- A system usually will have many components
- Fault in some component can lead to errors, which can lead to failure of the system
- Measuring how dependable a system is
  - Reliability
  - Availability
  - Safety

Dependability

- Reliability
  - How often does the system fail?
  - What is the conditional probability the system will work for the duration [0,t] given that it is working at time 0?
  - Measured by: MTTF (Mean Time To Failures), MTTR (Mean Time To Repair), MTBF (Mean Time Between Failures = MTTF + MTTR)
- Availability
  - How available the system is
  - What is the probability that the system is up at time t?
  - Usually measured by uptime (ex. 99%, max. downtime 5 hours in 1 year etc.)
- Safety
  - How safe the system is, even if it fails
  - Does it always maintain some safety property?

Some Observations

- A highly reliable system is also highly available
- A highly available system may or may not be highly reliable
  - Ex: If a system fails for 1 second every hour, it can still be considered highly available (99.97%) but not highly reliable
- The reliability of a system depends on the reliability of the components used to build the system
- Reliability/Availability can be of interest at different component levels
  - A memory chip
  - A disk controller with memory
  - A PC with disks
  - A cluster with a large number of PCs

Fault Tolerance

- The ability of a system to deliver desired services in spite of faults in its components
  - Can be full service (specified behavior in fault-free state)
    - Ex: A primary-backup server system to tolerate one server failure
  - Or a degraded service (deviate from specified behavior in fault free state, but in a pre-defined manner)
    - Ex: A web service with multiple load-balanced servers in the backend failing to meet its response time guarantees due to one backend server failure, but still giving service with slower response time

- Many modern distributed systems need to be highly available
  - Gmail
  - Facebook
  - Airlines/Railway reservation system
  - Many many others….
- But they also have very large number of components (machines, storage,….)
  - Chance of something failing at any time very high, even if individual MTBF is high
- How to build such systems in the presence of faults
  - But what are the different ways a system can fail?

Classification of Faults

- Based on component that failed
  - Program/process
  - Processor/machine
  - Memory
  - Link
  - Storage
  - Clock
  - …
- Too many possibilities
- But what matters finally is how does the system behave when faults happen

- Based on behavior of faulty component
  - Crash – just halts
  - Failstop – crash with additional conditions
  - Omission – fails to perform some steps
  - Byzantine – behaves arbitrarily
  - Timing – violates timing constraints
  - …

Types of Tolerance

- Masking – system always behaves as per specifications even in presence of faults
- Non-masking – system may violate specifications in presence of faults. Should at least behave in a well-defined manner

- A fault tolerant system should specify
  - Class of faults tolerated (Fault Model)
  - What tolerance is given from each class (Fault Tolerance)

Handling Faults

- Needs some redundancy
  - Hardware
  - Software
  - Time
  - Information
- Types of recovery
  - Forward error recovery
  - Backward error recovery

Some Building Blocks

- Primitive operations/components that are used as basic tool in the design of many fault tolerant systems
  - Building reliable storage from unreliable disks
    - RAID
    - Centralized networked storage
  - Reliable communication in the presence of unreliable links
    - Unicast, multicast, broadcast
  - Agreement/Consensus
  - Enforcing atomic actions across nodes
  - Checkpoint and recovery

Agreement Problems

Agreement Problems

- A set of n processes, m of which may be faulty
- Nonfaulty processes need to agree on some value(s) even in the presence of the faulty processes
- One of the most widely studied problem in distributed computing
  - Different problem variations studied under different models

Different Problem Variations

- Byzantine agreement (or Byzantine Generals problem)
  - One process x broadcasts a value v
    - All nonfaulty processes must agree on a common value (Agreement condition).
    - The agreed upon value must be v if x is nonfaulty (Validity condition)
- Consensus
  - Each process broadcasts its initial value
    - Satisfy agreement condition
    - If initial value of all nonfaulty processes is v, then the agreed upon value must be v

- Interactive Consistency
  - Each process i broadcasts its own value v_i
    - All nonfaulty processes agree on a common vector (v_1, $v_2, \ldots, v_n$)
    - If the i^{th} process is nonfaulty, then the i^{th} value in the vector agreed upon by nonfaulty processes must be v_i

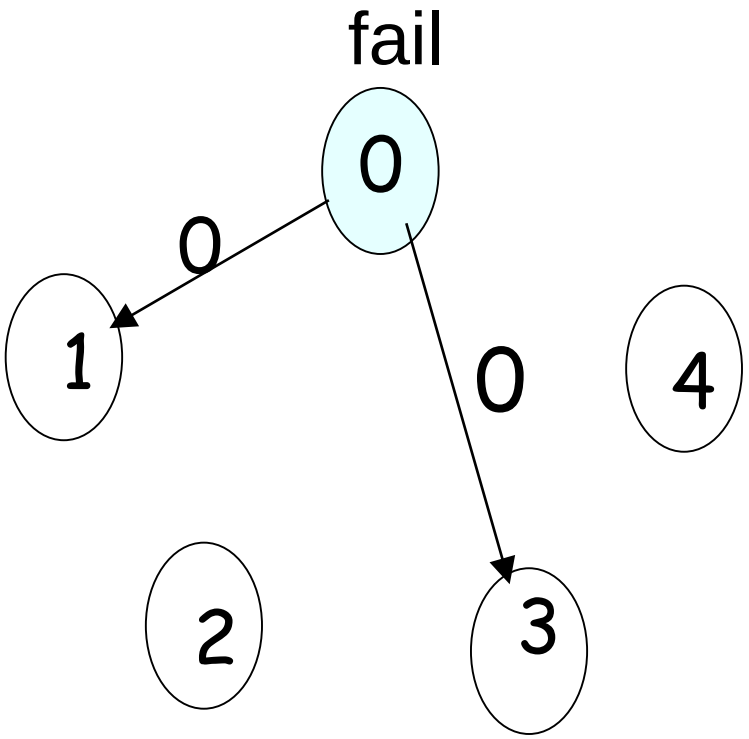
All three problems are equivalent, meaning that solution of any one of them can be used to solve the other two

Some Impossibility Results

- In the asynchronous model, consensus is impossible to achieve even with a single crash failure
- Consensus is impossible to achieve even with a single link failure, even in the synchronous model

Consensus Without Any Fault

- Completely connected topology
- Each processor
  - Broadcast its input to all processors
  - Decide on the minimum
- Only one round
- Does not work if there is a crash fault



Node 0 fails after broadcasting to nodes 1 and 3

1 and 3 decides 0, 2 and 4 decides 1 - violates Agreement Condition

Consensus With At Most m Crash Faults

- Synchronous, completely connected, reliable communication
- (m+1) rounds for each node
  - Round 1:
    - Broadcast own value to all other nodes
  - Round 2 to round m +1:
    - Broadcast any new received values
  - At the end of round m +1:
    - Decide on the minimum value received

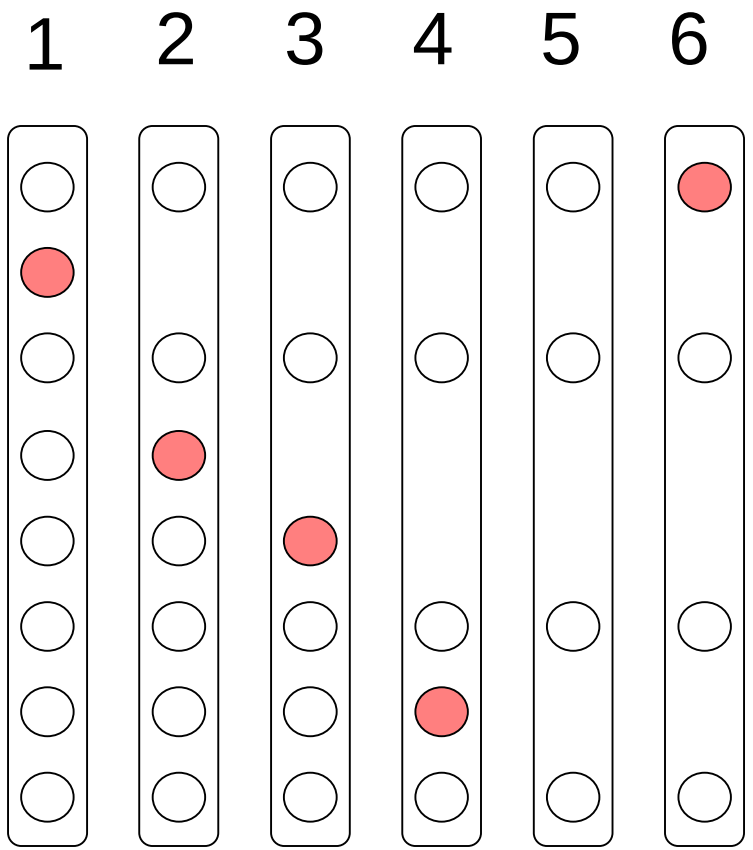
Key Idea

- In m +1 rounds, there is at least a round with no failed nodes
- At the end of this round, all non-faulty nodes have the values at all other non-faulty nodes
- For a faulty node
  - If it failed before sending its value to any other node, no one has its value
  - If it sent to some nodes before failing, at the end of the round with no failures, all non-faulty nodes will have its value
- However, it is not known where that no-failure round is, so have to go till m + 1 round

At least m+1 rounds are necessary to tolerate m crash faults by any consensus algorithm (lower bound)

5 failures,

6 rounds



No failure in round 5

Agreement under Byzantine Fault

- Total n processes, at most m of which can be faulty
- Synchronous, completely connected, reliable communication
- No solution possible if
  - $n < (3m + 1)$
- Needs at least (m+1) rounds of message exchange (lower bound result)

Lamport-Shostak-Pease Algorithm

- Oral messages – messages can be forged/changed in any manner, but the receiver always knows the sender
- Recursively defined algorithm:

OM(m), m > 0

- Source x broadcasts value to all processes
- Let v_i = value received by process i from source (0 if no value received). Process i acts as a new source and initiates OM(m -1), sending v_i to remaining (n - 2) processes
- For each i, j, i ≠ j, let v_j = value received by process i from process j in step 2 using O(m-1). Process i uses the value majority(v_1, v_2, …, v_{n-1})

OM(0)

- Source x broadcasts value to all processes
- Each process uses the value; if no value received, 0 is used

Time complexity = m+1 rounds

Message Complexity = $O(n^m)$

You can reduce message complexity to polynomial by increasing time

Atomic Commit

Atomic Actions and Commit Protocols

An action may have multiple subactions executed by different processes at different nodes of a distributed system

Atomic action: either all subactions are done or none are done (all-or-nothing property/ global atomicity property) as far as system state is concerned

Commit protocols – protocols for enforcing global atomicity property

Two-Phase Commit (2PC)

- Assumes the presence of write-ahead log at each process to recover from local crashes
- One process acts as coordinator, others are cohorts

Phase 1 (Prepare):

- Coordinator sends COMMIT_REQUEST to all cohorts
- Waits for replies from all cohorts
- On receiving a COMMIT_REQUEST, a cohort, if the local transaction is successful, writes Undo/redo logs in stable storage, and sends an AGREED message to the coordinator. Otherwise, sends an ABORT

Phase 2 (Commit):

- If all cohorts reply AGREED, coordinator writes commit record into the log, then sends COMMIT to all cohorts. If at least one cohort has replied ABORT (or timeout), coordinator writes abort record in log and sends ABORT to all. Coordinator then waits for ACK from all cohorts. If ACK is not received within timeout period, resend. If all ACKs are received, coordinator writes complete to log
- On receiving a COMMIT, a cohort releases all resources/locks, and sends an ACK to coordinator
- On receiving an ABORT, a cohort will undo the transaction using Undo log, releases all resources/locks, and sends an ACK

- Ensures global atomicity (either all processes commit or all of them aborts)
- Resilient to arbitrary no. of crash failures
  - Think of different failure scenarios
    - Coordinator fails at different steps
    - Cohort fails at different steps
    - Their failures are independent, so any no. can fail simultaneously
  - Whenever you think of crash of a process (coordinator or other), think also of what happens when it recovers
- Problem: Two-phase Commit is a blocking protocol
  - Crash of coordinator can block all processes until coordinator recovers

- Example of a failure and recovery scenario
  - Failure scenario: suppose all cohorts have replied with AGREED in Phase 1, coordinator has written commit record in its log, and then fail after sending zero or more (but not all) COMMIT messages
    - The cohorts who get the COMMIT will commit, and reply with ACK. Coordinator does not get it as it is still crashed.
    - The cohorts who do not get the COMMIT are blocked until coordinator recovers
  - Recovery: The coordinator recovers after some time
    - The coordinator checks its log, sees the commit record and thereby knows that all cohorts must have replied AGREED
    - It does not see the complete record in log, so knows all ACKs have not been received before crash
    - Sends COMMIT to all cohorts again. Cohorts take action as in protocol and two-phase commit completes if no further crash.

Non-Blocking Commit Protocols

- Three-phase Commit (3PC)
  - Assumptions
    - No network partitioning
    - Less than K processes can fail at any time
    - At least one process stays up at all time
  - Drawbacks:
    - Higher overheads
    - Assumptions may not be satisfied in practice
  - Hardly used in practice
- Two-Phase Commit used widely in practice as probability of blocking is low in practical systems

Checkpointing & Recovery

Checkpointing & Rollback Recovery

- Forward error recovery – assess damage due to faults exactly and repair the erroneous part of the system state
  - Less overhead but hard to assess effect of faults exactly in general
- Backward error recovery – on a fault, restore system state to a previous known error-free state and restart from there
  - Costlier, but more general

Checkpoint and Rollback Recovery – a form of backward error recovery

Checkpoint :

- Local checkpoint – local state of a process saved in **stable storage** for possible rollback on a fault
- Global checkpoint – collection of local checkpoints, one from each process

Consistent and Strongly Consistent Global Checkpoint – similar to consistent and strongly consistent global state respectively (Also called "recovery line")

Main Idea: On recovery after failure, restart the system from an earlier consistent state

Widely used for long running computational tasks

Orphan message – a message whose receive is recorded in some local checkpoint of a global checkpoint but send is not recorded in any local checkpoint in that global checkpoint (Note : A consistent global checkpoint cannot have an orphan message)

Lost message – a message whose send is recorded but receive is not in a global checkpoint

Are lost messages a problem??

- Not if unreliable channels assumed (since messages can be lost anyway)
- If reliable channels assumed, need to handle this properly! Cannot lose messages !

Performance Measures

- During fault-free operation
  - Checkpointing time
  - Space for storing checkpoints and messages (if needed)
- In case of a fault
  - Recovery time (time to establish recovery line)
  - Extent of rollback (how far in the past did we roll back? how much computation is lost?)
  - Is output commit problem handled? (if an output was sent out before the fault, say cash dispensed at a teller m/c, it should not be resent after restarting after the fault)

Some Parameters that Affect Performance

- Checkpoint interval (time between two successive checkpoints)
- Number of processes
- Communication pattern of the application
- Fault frequency
- Nature of stable storage

Classification

- Asynchronous/Uncoordinated
  - Every process takes local checkpoint independently
  - To recover from a fault in one process, all processes coordinate to find a consistent global checkpoint from their local checkpoints
  - Very low fault-free overhead, recovery overhead is high
  - Domino effect possible (no consistent global checkpoint exist, so all processes have to restart from scratch)
  - Higher space requirements, as all local checkpoints need to be kept
  - Good for systems where fault is rare and inter-process communication is not too high (less chance of domino effect)

- Synchronous/Coordinated
  - All processes coordinate to take a consistent global checkpoint
  - During recovery, every process just rolls back to its last local checkpoint independently
  - Low recovery overhead, but high checkpointing overhead
  - No domino effect possible
  - Low space requirement, since only last checkpoint needs to be stored at each process

- Communication Induced
  - Synchronize checkpointing with communication, since message send/receive is the fundamental cause of inconsistency in global checkpoint
  - Ex. : take local checkpoint right after every send! Last local checkpoint at each process is always consistent. But too costly
  - Many variations are there, more efficient than the above.

- Message logging
  - Take coordinated or uncoordinated checkpoint, and then log (in stable storage) all messages received since the last checkpoint
  - On recovery, only the recovering process goes back to its last checkpoint, and then replays messages from the log appropriately until it reaches the state right before the fault
  - Only class that can handle output commit problem!

Some Checkpointing & Recovery Algorithms

- A large number of algorithms exist for each of the classes
- We will look at one synchronous algorithm from text
  - See Koo-Toueg's algorithm