Fault Tolerance in Distributed Systems

Introduction

- A fault-tolerant system
  - masks failures, i.e. continues to operate despite failures
  - exhibits well defined failure behavior, i.e. facilitates recovery but it may not continue operating
- Failure types
  - process deaths
  - machine crashes
  - network failures: link failures, network partitions, message losses
- Atomic actions (transactions)
  - process executing actions does not communicate/affected by other processes while executing these actions
- Global atomicity
**Basic Mechanisms for Recovery**

- Two protocols to restore/recover the updated objects in a consistent state are:
  - **Write-ahead-log protocol**
    - Before updating an object, write an UNDO log to stable storage.
    - Update the object.
    - Before committing the update, write a REDO log to stable storage.
  - **Shadow page protocol**
    - Make a “shadow” copy of the object in stable storage.
    - Update the object.
    - Before committing, update the shadow copy of the object.
- On recovery, either use the UNDO/REDO logs or the shadow page to restore an object.

**Distributed Commit & the Generals Paradox**

- **The generals paradox**
  - Two generals communicating via messengers need to coordinate to capture a hill; in order to succeed both must attack.
  - There is no protocol to solve this problem with a bounded number of messages.
- **Proof (by contradiction)**
  - Consider a message exchange protocol P with min #messages.
  - Consider the last message exchanged M does not reach one general.
  - Is M really necessary for the generals to capture the hill?
    - If yes, the generals fail to capture the hill.
    - If no, then P is not a protocol with min #msgs.
2-Phase Commit

- System consists of a coordinator and cohorts
  - Coordinator sends Commit_Request, Commit, and Abort messages
  - Cohorts send Agreed to commit and Abort messages
- Coordinator
  - sends Commit_Request messages to all cohorts and waits for their replies
  - if all cohorts replied Agreed, then coordinator writes COMMIT log, sends Commit messages to all cohorts, and waits for Acknowledgement
  - if any cohort replied Abort, it sends Abort to all cohorts and Aborts
  - when all cohorts acknowledge Commit msg, it writes COMPLETE log
  - if it does not receive a reply it resends the request message

2-Phase Commit

- Cohorts
  - upon receiving Commit_Request reply with Agreed or Abort message based on their local state, and wait for a reply from coordinator
  - if coordinator replies with Commit, it commits
  - if coordinator replies with Abort, cohort undoes updates, and aborts
  - in either case, it sends reply(ACK) to coordinator

- 2PC is a blocking protocol
  - if coordinator or cohort fails or a message is lost, transaction may neither Aborts nor Commits
Non-Blocking Commit

- Independent recovery
  - if recovering sites can determine the outcome of a transaction based solely on their local state

- Protocols
  - are modeled by finite state machines with arcs labeled with conditions/actions.
  - synchronous within 1 state transition if the distance between the number of transitions between any two sites is $\leq 1$

- Concurrency set $C(x)$ of a state $x$ at a node
  - set of states any other node can be in, while that node is in state $x$

- Sender set $S(x)$ of a state $x$ at a node
  - set of nodes that can send the messages which can be received in state $x$

Theorem A Commit protocol is resilient (i.e. does not block) under an arbitrary single failure if the concurrency set of any state does not contain conflicting final states, under the independent recovery assumption.

To make a protocol resilient we may need to introduce new states so that the above theorem applies.

Theorem Under independent recovery
- there is no resilient commit protocol for more than 1 site failures
- no commit protocol is resilient to network partitioning with message losses
- no commit protocol is resilient under multiple network partitions
Non-Blocking Commit: 3-Phase-Commit

- Adding failure and timeout transitions to a commit protocol
  - Rule 1: for every non-final state \( x \) with concurrency set \( C(x) \), if \( C(x) \) contains a Commit final state \( y \) add a failure transition from \( x \) to \( y \)
  - Rule 2: for every non-final state \( x \), if a node \( j \) in the sender set of \( x \) has a failure transition to a commit/abort final state, add a timeout transition from \( x \) to commit/abort final state
  - these two rules are sufficient for resilient commit protocols for single site failures
  - Starting with the two phase commit by adding “buffer” states between the waiting and commit states we obtain the three phase commit protocol; failure/timeout transitions are added according to the rules 1 and 2 above
3-phase commit protocol
Cohort i (i=2,3,...,n)

Commit_Request
msg received
Agreed msg sent to Coordinator

Commit_Request
msg received
Abort msg sent to Coordinator

F,T

F,T

Commit_Request
msg received
Abort msg received from Coordinator

Send Ack msg to Coordinator

Commit msg received from Coordinator

Abort msg received from Coordinator

T- Timeout Transition
F- Failure Transition
F,T – Failure/Timeout Transition

Gifford’s Static Voting Protocol

Voting protocols provide a method for managing replicated objects in way that is resilient to a richer set of failures

Gifford’s Static Voting protocol
- there is a read and a write quorum
- every site has a lock manager that is used to lock an object X
- in order for a site to cast its votes, it needs to obtain a lock from its lock manager;
- every site for each replica X it has, it maintains a version# for X and its own number of votes
Gifford’s Static Voting Protocol

A site i wishing to read/write X
- requests a lock from its local lock manager
- sends vote requests to all other sites
- collects the (Votes, Version#) from all the replying sites
- determines if it has a read/write quorum
- ensures its local copy is current
- performs all its local operations on X and then updates, if it got a write quorum, all (current) replicas that replied
- releases its local lock, and
- sends release_lock msgs to all the nodes that replied

Gifford’s Static Voting Protocol

a site j
- upon receiving a vote_request
  - obtains a local lock, and
  - sends its #votes# and version# to requestor
- upon receiving a release_lock, it releases its local lock
- determining read/write quorums
  - read quorum: #votes received is >= read quorum R
  - write quorum: #votes received from nodes with most current replica >= write quorum W
  - R+W > total #votes and W > total #votes / 2
- fault tolerance and performance obtained with Gifford’s static voting protocol depends on the assignment of votes to nodes, the availability of these nodes, and the size R and W of the read/write quorums
Dynamic Voting Protocols

- Network partitions and other failures can hurt fault-tolerance of static voting
- Dynamic voting can boost fault-tolerance by adapting
  - the number of votes assigned to various nodes
  - the set of nodes that can form read/write quorums
- Partition graph
  - directed graph whose
    - vertices denote the various partitions of the nodes of the system
    - edges denote how partitions are formed (via merging and splitting)

Majority Based Dynamic Voting Protocol

- Idea for the Jajodia-Mutchler’s protocol
  - modify the read/write quorums so that sites in a distinguished network partition can still operate
  - A network partition is distinguished if it contains a majority of the nodes that participated in the last update operation
- Every site i maintains (with respect to its view of the world)
  - version number $V_N_i$ = #updates performed on its local copy of the object
  - #replicas updated $R_{U_i}$ in most recent update according to site i
  - distinguished sites list $D_{Si} = a$ set of node ids
    - if $R_{U_i} = 3$ then $D_{Si}=$set of all participating nodes in most recent update
    - if $R_{U_i}$ is even then $D_{Si}=$ID of highest priority node among those participating in most recent update
    - else $D_{Si}=$null
**Majority Based Dynamic Voting Protocol**

- Nodes that want to read/write a replica attempt to obtain a read/write quorum as in Gifford, except that
  - if a node determines that is not in the distinguished partition, then
    - it aborts and releases the locks/voters
  - otherwise, if the node updates the replica object, then
    - it updates its VNi, RUi, and DSi, and then
    - sends a Commit message together with these updated values to all nodes participated in the update

**Majority Based Dynamic Voting Protocol**

- Consider a site I requesting votes
  - let P the set of responding nodes the vote request
  - let M be the highest version number among the nodes responded
  - let Q be the set of nodes that responded and have the most current replica
  - let N be the RU value of any node in Q
  - Site I is in the distinguished partition if
    - Size of(Q) > N/2
    - or DSj is in Q for some (any) j in Q
    - or N=3 and P contains at least two of the nodes in DSj, j in Q
  - Site I updates VNi, RUi, DSi, as follows
    - sets VNi to M +1
    - sets RUi to size of(P), and
    - sets DSi to P if RUi=3, to the ID of highest priority node in Q if RUi is even, and to null otherwise
Autonomous Vote Re-Assignment

- Vote re-assignment is another method for increasing the fault-tolerance of voting protocols, by changing the #votes of a select group of nodes in anticipation of future failures.
- It can be node in a consensus or an autonomous manner.
- We consider an autonomous vote re-assignment protocol.
  - Each node i maintains three vectors:
    - $V_i[j] =$ #votes of node j according to node i
    - $Ni[j] =$ version of #votes for node j according to node i
    - $v_i[j] =$ #votes node j reports it has in response to a request by node i.
  - Protocol has three sub-protocols:
    - Vote increasing
    - Vote decreasing
    - Vote collecting

Vote Increasing Sub-Protocol

- When node I wishes to increase its #votes:
  - Selects its new #votes $X$.
  - Sends its $V_i$ and $Ni$ vectors, together with its new #votes $X$, to all other nodes it can reach.
  - Waits until it gets a majority using the vote collecting protocol.
  - If it gets a majority, then it sets:
    - $V(i)[i] = X$
    - $Ni[i] = Ni[i]+1$

- Each node j upon receiving $V_i$, $Ni$, and new #votes $X$ for I:
  - Sets $Vj[i] = X$
  - Sets $Nj[i] = Ni[i]+1$
**Vote Decreasing Sub-Protocol**

- When a node $I$ wishes to decrease its #votes
  - selects its new #votes $X$
  - sets $V[i] = X$
  - sets $N[i] = N[i] + 1$
  - and then sends the vectors $V_i$ and $N_i$ to all other nodes

- A node $j$ receiving $V_i$ and $N_i$ from node $i$
  - sets $V[j] = V[i]$
  - sets $N[j] = N[i]$

- note that vote reduction does not endanger mutual exclusion, thus vote collection to determine majority is not needed

---

**Vote Collecting Protocol**

- Vote collection is done anytime agreement is needed, i.e.
  - when a node $i$ wants to read, write, or re-assign its #votes

- Node $i$ upon receiving a reply ($V_j$, $N_j$) to a vote request
  - sets $v[i][j] = V[j][j]$, and if
    - $V[j][j] > V[i][j]$ (node $i$ missed a vote increase for node $j$)
    - or $V[j][j] < V[i][j]$ and $N[j][j] > N[i][j]$ (node $i$ missed a vote decrease by $j$),
  - then sets $V[i][j] = V[j][j]$ and $N[i][j] = N[j][j]$

- If node $i$ does not receive a reply from a node $j$ then it updates its $v_i$, $V_i$, and $N_i$ for node $j$ using the information from the vectors $V_k$ and $N_k$ for node $j$ where node $k$ has the most current information for node $j$

- Node $i$ collects a majority if #votes received is $> \text{half the sum of the entries of } v_i \text{ over all nodes}$
**Vote Increasing Policies**

- **the overthrow technique**
  - upon detecting failure of a node i, a node j is selected to increase its voting power
  - node j increases its voting power by at least twice the voting power of the failed node i
    - Otherwise mutex is in jeopardy

- **the alliance technique**
  - nodes in the majority collectively increase their #votes by some collective mechanism (all get some extra votes)
  - the increase in the #votes is always at least twice the #votes of the failed node(s)

**Failure Resilient Processes**

- **A process is resilient if it masks failures and guarantees progress despite certain number of failures**

- **approaches for building resilient processes**
  - Backup processes and replicated execution

- **Backup processes**
  - primary process and one or more backup processes
  - state of primary is checkpointed periodically and made accessible to the backup processes
  - backup process needs to detect failure of primary and before it starts execution it may need to perform some re-computation since the last checkpoint
**Replicated Execution**

- Several processes execute the same program concurrently

**Replicated execution**

- increases
  - reliability by taking a majority consensus among the results of all the processes
  - availability, since service is available as long as one process continues executing
- requires extra resources
- has problems with non-idempotent operations such as
  - use of random number generators
  - cooperative computations via message exchange

**Reliable Communication**

- Consider a system of cooperating processes that communicate by exchanging messages, such as maintaining replicated data, where all processes must have the same view of the system
- this requirement can be met if atomic multicast is supported
  - the messages are received in the same order at each process
  - each message is either received at every process or at none of them
- **Note that in an atomic multicast**
  - We have a total ordering of all messages for all processes
  - but such an ordering may not be causal
Birman-Joseph Atomic Broadcast Protocol

- Each process has an associated message priority queue
  - messages are buffered in the queue before delivered to the process
  - messages have a priority# and can be in two states
    - deliverable and undeliverable
  - a message is delivered to the process when it is deliverable and has the smallest priority# among all buffered messages
  - messages are retained even after they are delivered
- the protocol has two phases and a fault handler

Phase I

- sender sends a message together with the ids of the receivers to all the processes in the multicast group
- a receiver, upon receiving the message,
  - assigns it a priority# (highest among all its buffered messages)
  - marks it undeliverable and buffers it in the queue
  - informs the sender of the assigned message priority#
**Birman-Joseph Atomic Broadcast Protocol**

**Phase II**
- sender, upon receiving the responses from everybody in the multicast group that is still operational
  - sets the final priority# of the message as the maximum among all priority# assigned by the receivers
  - multicasts the final priority# to the multicast group
- receiver, upon receiving final priority# of message
  - assigns the priority# to the corresponding message
  - marks the message as deliverable

**Birman-Joseph’s Atomic Broadcast – Fault Handler**

- If receiver detects that the sender for an undeliverable message has failed then it attempts to complete the protocol as coordinator by performing the following steps
  - Step 1
    - asks the nodes in message’s multicast group about the statues of the message
    - a receiver can respond either with the status and priority# of the message or that it has not received the message
  - Step 2
    - if the message was marked deliverable at any node in the multicast group, Phase II is executed again with the “coordinator” multicasting the message’s final priority#
    - otherwise, the “coordinator” reinitiates the protocol from Phase I