Storage Systems

NPTEL Course

Jan 2012

(Lecture 31)

K. Gopinath

Indian Institute of Science

Multicast Oscillatory Behaviour

- Nodes experience disturbances eg. Java gc pauses, Linux sched delays, flushing data to disk
- Prevents nodes from forwarding packets eg. when appl thread does not respond or when packets do not reach node because of a link problem.
- For a while, root continues sending, so incoming packets from the upstream link fill node's buffers.
- Flow control causes node's parent node to stop sending, which in turn causes its buffers to fill up.
- If node's disturbance persists, then eventually all buffers on path from root to the node become full, and root's sending throughput drops to zero
- In large trees (10K-60K nodes in cloud), when each node is disturbed for one sec/hour on average, throughput degradation (up to 90%) occurs even if message loss is negligible.

FLP/CAP Related Problems

- Distributed Consensus: FLP Impossibility result
  - Distributed Locking/Synchronization
  - Distr Commit in clustered/distr fs and db
    - Slightly similar: Waitfree synchronization
- Let us consider the state of art in current large scale storage systems:
  - Distr locking, synch, commit problems exist
  - How are they being handled? What guarantees are being given?

Storage APIs

- POSIX: Read (fd, buffer, count)
  - Partial writes to a file OK (appends, overwrites, etc)
  - mmap avlbl
- NFS: Read (fd, ***offset***, buffer, count)
  - Partial writes and mmap avlbl
- Amazon S3: “storage” service
  - Key Value store; no features like partial write or mmap!
- ZooKeeper: hierarchical “file”-like service
  - In memory tree-based info for distributed coordination
  - Replicated
  - Provides primitives to construct more complex services
    - Synchronization, group membership

S3 Interface: Key Value Store

- Amazon S3 stores data in named buckets
  - Each bucket is a flat namespace, containing keys associated with objects (but not another bucket)
  - Max obj size 5GB. Partial writes to objects not allowed (must be uploaded full), but partial reads OK
- Storage API
  - create bucket
  - put bucket, key, object
  - get bucket, key
  - delete bucket, key
  - delete bucket
  - list keys in bucket
  - list all buckets

ZooKeeper

- Tree-based info (“filesystem”)
- Fast and simple
- each node stores one or more pieces of info (“file”)
- very simple programming interface:
    - create: creates a node at a location in the tree
    - delete: deletes a node
    - exists: tests if a node exists at a location
    - get/set data: reads/writes the data from a node
    - get children: retrieves a list of children of a node
    - Sync: waits for data to be propagated

Eventual Consistency

- S3 model: **When no updates occur for a long period of time**, eventually all updates will propagate through the system and all the replicas will be consistent
  - Often called BASE: Basically Available, Soft-state and Eventually Consistent!
  - Contrast with ACID
- Zookeeper consistency model:
  - The clients view of the system is guaranteed to be up-to-date ***within a certain time bound***

ZooKeeper Consistency model

- guarantees:
  - Sequential Consistency - Updates from a client will be applied in the order that they were sent.
  - Atomicity - Updates succeed or fail. No partial results.
  - Single System Image - A client will see the same view of the service regardless of the server that it connects to.
  - Reliability - Once an update has been applied, it will persist from then until a client overwrites the update.
  - Timeliness - The clients' view of the system is guaranteed to be up-to-date *within a certain time bound*.

ACID vs. BASE

- ACID
  - Strong consistency, Isolation, Focus on “commit”
  - Availability?
  - Conservative (pessimistic)
  - Nested transactions
  - Difficult System evolution
- BASE
  - Weak consistency: stale data OK
  - Availability first, Best effort, Approx answers OK
  - Aggressive (optimistic)
  - Simpler and Faster
  - Easier System evolution

Commit Protocols

- Abstract problem related to consistency: commit or consensus protocols
- Atomic Commitment (AC) and Consensus: both require fault tolerant agreement among processes

  AC:
- AC1: No two processes reach different decisions.
- AC2: Commit is decided only if all votes are Yes.
- AC3: If there are no failures and all votes are Yes, then all processes decide to Commit.
- AC4: If all existing failures are repaired and no new failures occur for a sufficiently long period of time, then all processes will reach a decision.
- No Blocking: All correct processes reach a decision: Unrealizable! (General's paradox)

Consensus

- Agreement (A) All non-faulty processes reach the same decision
- Validity (V) If all non-faulty processes' votes are *Yes,* they will all decide to Commit; if all non-faulty processes' votes are *No,* they will all decide to Abort
- Weak Validity (WV) If there are no failures, V holds
- Very Weak Validity (VWV) Both Commit and Abort are possible decision values: i.e. there is an execution in which correct processes decide to Commit and an execution in which correct processes decide to Abort
- Satisfaction of A and V: consensus problem.
- Satisfaction of A and WV: weak consensus
- Satisfaction of A and VWV: very weak consensus

Relation Betw AC and Consensus

- Differences betw AC and diff versions of consensus concern
- the decisions reached by faulty processes; and
- the strength of the conditions required

AC attainable only under the assumption that process

failures are benign

Can prove

- AC 2,3,4 *imply* WV but not the converse
- With “no-catastrophe” axiom (NC): all failures repaired and no new failures for a sufficient period of time, then AC1, AC4 and NC *imply* A

AC conditions stronger than WV, assuming NC