Practical Replication

Purposes of Replication
- Improve Availability
  - Replicated databases can be accessed even if several replicas are unavailable
- Improve Performance
  - Replicas can be geographically diverse, with closest replica serving each client

Problems with Replication
- Consistency of the replicated data
  - Many applications require consistency regardless of which replica is read from or inserted into
  - Consistency is expensive
  - Some replication schemes will reduce update availability
  - Others require reconciliation after inconsistency occurs
  - Performance may suffer as agreement across replicas may be necessary

The Costs and Limits of Availability for Replicated Services
- Consistency vs. Availability
  - Many applications don't need strong consistency
  - Can specify a maximum deviation
  - Consistency don't need to be sacrificed during normal operation
  - Only perform tradeoff when failure occurs
  - Typically two choices of consistency
    - Strong consistency
      - Low availability, high data accuracy
      - Weak consistency
    - High availability, low accuracy (lots of conflicts and stale access)
- Continuous Consistency Model
  - A spectrum of different levels of consistency
  - Dynamically adapt consistency bounds in response to environmental changes

Continuous Consistency Model

Metrics of Consistency
- Three categories of errors in consistency at a replica
  - Numerical error
    - The total number of writes accepted by the system but not seen by the replica
  - Staleness
    - Difference between current time and the acceptance time of the oldest write not seen locally
  - Order error
    - Number of writes that have not established their commit order at the local replica
Example of Numerical and Order Error

![Example Diagram]

### Deriving tight upper bound on availability
- Want to derive a tight upper bound on the availability based on a given level of consistency, workload, and faultload.
- \( \text{Avail}_{\text{total}} \leq f(\text{consistency, workload, faultload}) \)
- Upper bound helps evaluate existing consistency protocols
- Reveal inherent impact of consistency on availability
- Optimize existing consistency protocols
- Questions:
  - Must determine which write to accept or reject
  - Accepting all writes that do not violate consistency may preclude acceptance of a larger number of writes in the future
  - Determine when and where to propagate writes
  - Write propagation decreases numerical error but can increase order error
  - Must decide serialization order
  - Can affect the order error

### Upper bound as a function of numerical error and staleness
- Questions on write propagation
  - When and where to propagate writes
  - Simply propagate writes to all replicas whenever possible – Aggressive write propagation
  - Always help reduce both numerical error and staleness
- Questions on write acceptance
  - Must perform an exhaustive search on all possible sets of accepted writes
  - To maximize availability and ensure numerical and staleness bounds are not violated
  - Search space can be reduced by collapsing all writes in an interval to a single logical write
  - Due to Aggressive write propagation

### Upper bound as a function of order error
- To commit a write, a replica must see all preceding writes in the global serialization order
- Must determine the global serialization order
- Factorial number of serialization orders
  - Search space can be reduced
    - Causal order
      - Serialization orders compatible with causal order
    - Cluster order
      - Writes accepted by the same partition during a particular interval cluster together

### Serialization order
- Example:
  - Suppose Replica 1 receives transaction \( W_1 \) and \( W_2 \), and Replica 2 receives \( W_3 \) and \( W_4 \)
  - Causal:
    - \( S = W_1, W_2, W_3, W_4 \) better than \( S' = W_2, W_1, W_3, W_4 \)
    - Whenever \( W_1 \) can be committed using \( S' \), the replica must have already seen \( W_1 \) and thus can also commit \( W_2 \) in \( S \). The same is true for \( W_3 \) and \( W_4 \)
  - Cluster:
    - Only 2 possible clusters
      - \( S = W_1, W_2, W_3, W_4 \) and \( W_1, W_2, W_3, W_4 \)
    - Intuition is that it does not expedite write commitment on any replica if the writes accepted by the same partition during a particular interval are allowed to split into multiple sections in the serialization order
    - Cluster has smallest search space

### What can we get from this?
- Modify an existing protocol with ideas from proof
  - Each replica ensure that the error bound on other replicas are not violated
  - Added aggressive write propagation
- Analyze other protocols for order error
  - Primary copy protocol
    - A write is committed when it reaches the primary replica
  - Golding’s algorithm
    - Each replica maintains a version vector to determine whether it has seen all writes
  - Pulls in writes from other replicas to advance version vector
  - Voting
    - Order is determined by voting of members
- Is there anything else other than Aggressive Write Propagation that we can get from this proof?
Numerical Error Bound

- As predicted, aggressive write propagation improves availability.
- Also increases the number of messages required.
- Removes the optimization of combining multiple updates to amortize communication costs.
- Packet header overhead, packet boundaries, ramping up to the bottleneck bandwidth in TCP.

Order Error Bound

- Aggressive Voting also performs well.
- Base voting is awful.
- Lazy replication can cause each replica to cast a vote for a different uncommitted write.
- Each replica must collect votes from all replicas to determine the winner and any unknown vote can be the deciding one.
- Aggressive ensures most votes for the same uncommitted write.
- Only need to contact a subset of nodes.

Effects of Replication Scale

- Adding more replicas.
- Reduces network failure rate.
- Increases replica rejection rate.
- Availability = \((1 - \text{Network Failure Rate}) \times (1 - \text{Rejection Rate})\).

The Dangers of Replication and a Solution

- Replication works well with a few nodes.
- Limited deadlocks and reconciliation needed.
- Does not scale well or handle mobile nodes that are normally disconnected.
- Cubic growth of deadlock and reconciliation rates predicted in this paper.
- Is this a fundamental limitation?
- Eventually reaches system delusion.
- Database is inconsistent and there is no obvious way to repair it.
- What about mobile nodes?
- Does replication currently work well with nodes that can be disconnected?

How does replication models affect deadlock/reconciliation rates

- Models to propagate updates to replica:
  - Eager replication:
    - Updates applied to all replicas of an object as part of original transaction.
  - Lazy replication:
    - One replica is updated by the original transaction.
    - Updates to other replicas propagate asynchronously as separate transactions.

- Models to regulate replica updates:
  - Group:
    - Any node with a copy of the data can update it.
  - Master:
    - Each object has a master node.
    - Only master can update the primary copy.

Eager Replication

- Updates all replicas in same transaction.
- No serialization anomalies, no need for reconciliation.
- Not an option for mobile systems.
- Updates may fail even if all nodes are connected all the time.
- When replicated, deadlock rate grows cubic to the rate number of nodes.
- Each node must do its own work and also apply updates generated by other nodes.
- Probability of a wait also increases.

\[
\text{TPS} \times \text{Actions} \times \text{Actions} \times \text{Nodes} \quad + \quad \text{DB \_Size} / 4
\]

- Deadlocks can be removed if used with an object-master approach.
- Lower throughput due to synchronous updates.
**Lazy Group Replication**
- Any node can update any local data
- Updates are propagated asynchronously in separate transactions
- Timestamps are used to detect and reconcile updates
  - Each object carries the timestamp of its most recent update
  - Each replica update carries the new value and is tagged the old object timestamp
  - Receiving replica tests if local timestamp and the update's old timestamp are equal
  - If so, update is safe, local timestamp advances to the new transaction timestamp
  - Else, update may be dangerous, and requires reconciliation on the transaction

**Lazy Master Replication**
- Updates are propagated asynchronously in separate transactions
- Only object master can update object
- No reconciliation required
- Deadlock possible
  \[
  \frac{\text{TPS} \times \text{Nodes}^3 \times \text{Action Time} \times \text{Actions}^3}{4 \times \text{DB Size}}
  \]
- Not appropriate for mobile applications
- Requires atomic transaction with the owner

**Non-Transactional Schemes**
- Let’s be less ambitious and reduce the domain
  - Abandon serializability for convergence
  - Add timestamps to each update
    - Lotus Notes approach:
      - If update has a greater timestamp than current, replace current
      - Else, discard update
  - System works if updates are commutative
    - Value is completely replaced
    - Adding or subtracting constants
    - May not even need timestamp

**Two-tier system**
- Two node types
  - Mobile Nodes
  - Often disconnected
  - May originate tentative transactions
  - Base nodes
  - Always connected
- Two version types
  - Master Version
    - Most recent value received from object master
  - Tentative Version
    - Local version and may be updated by tentative transactions

**Two-tier system**
- Base Transaction
  - Work on master data and produces new master data
  - Involved with at most one mobile node, and several base nodes
- Tentative Transaction
  - Work on local tentative data
  - Produces tentative version and a base transaction to be run later on the base nodes
  - Base transaction generated by tentative transaction may fail or produce different results
  - Based on a user specified acceptance criteria
    - E.g. The bank balance must not go negative
Two-tier system

- If tentative transaction fails
  - Originating node informed of failure
  - Similar to lazy-group replication except
    - Master database is always converged
  - Originating node need to only contact a base node to discover whether the tentative transaction is acceptable

Example

- When Mobile node connects
  - Discard tentative object version since it will be soon refreshed
  - Send its master object updates
    - Objects that the mobile node is master
  - Send all tentative transactions
  - Accept replica updates from the base node
  - Accept notice of success or failure of each tentative transaction

Example

- On host
  - Send delayed replica updates to mobile node
  - Accepts delayed mobile-mastered objects
  - Accepts list of tentative transactions with acceptance criteria
  - After base node commits, propagate update to other replicas
  - Converge mobile node state with base state

Two-tier system

- Does the two-tier system solve the scalability of replication problem
  - Yes, but only if we can restrict the domain
- Can we do better?
  - Or is this a fundamental problem that can’t be solved entirely?