Hash-History Approach for Reconciling Mutual Inconsistency in Optimistic Replication

B. Hoon Kang, Robert Wilensky and John Kubiatowicz  
CS Division, UC Berkeley

Optimistic Replication

- Optimistic replication has been widely used in distributed systems to achieve increased availability and performance.
- The definition of “optimistic”:
  - allows the replica to be updated in any place
  - and later converges to a consistent state by reconciling with each other the updates that each site has accrued independently.
- The reconciliation process between replicas:
  - needs a mechanism to determine the version dominance (i.e. which version is newer) or the update-conflict.

Previous Approaches: Version Vectors

Site A initiates reconciliation with site B and site A merges the state (i.e. version) V1 and V2 by deciding the ordering as b1 and a1.
- If operations (a1,b1) are commutative, the outcome would be the same.
- Later, site C initiates reconciliation with site A and merges the version V4 and V5 by deciding the ordering as b1, a1 and c1.

Our Approach: Hash History

- Hash History Approach:
  - Each site keeps a record of the hash of each version
  - The sites exchange the list of hashes in reconciling replicas
  - The most recent common ancestor version can be found, if no version dominates
  - Useful bins in a subsequent merging

- Scalable to thousands of sites
- Hash lists grow in proportion to number of update instances not number of sites
- The number of update instances can be bounded by flushing out obsolete hashes
- Simple to maintain
  - No need to track which site made changes
  - Only track what are changes there have been so far
- No need to naming the sites
  - Suitable for ad-hoc peer-to-peer networks

Hash History Pruning

- Aging with Loosely Synchronized Clocks
  - The classical techniques
    - The global-cutoff timestamp (Lynch et al) and the acknowledgement-timestamp (Golding et al)
    - Source: this method fundamentally requires to track the committed state per each site.
    - Hence it would not scale to thousands of sites.
  - We chose to use the simplistic aging method based on roughly synchronized timestamp.

- Highly Shareable Archived Hash History
  - Unlike version vectors, the hash-history for the shared data can be easily shared among many sites
  - Since it does not contain site-specific information rather it contains the histories of the shared object.
  - One can easily convinced that archiving the old history at one of the primary sites should be good enough to handle the special case: the version that belongs to the obsolete (pruned) hash-history can be mistakenly considered as a new version.

Simulation Result

- We ran the simulation based on the traces that we collected from the cvs-logs of some active projects on sourceforge.net.

Correctness Test

- We implemented the dynamic version vector scheme
  - To compare with the hash-history mechanism
  - To see if there is any case that hash-history mechanism would determine the version dominance differently
- The version vector and hash history returns same results.
- We made the merge procedure To Produce unique output
  - so that there will be no case when two different series of delta produce the same result.
  - Since version vector pessimistically assume such case, the results was exactly the same as in the table above.

Effectiveness Analysis

- We use a deterministic merge policy to simulate a deterministic merge behavior
  - so that two different schedules of deltas produce the same output
  - When two hash histories are merged,
    - we have the merge process to automatically pick the one with higher timestamp as a new version.
    - And we enabled this new version’s hash so that it can be distinguishable from its parents.
  - Surprisingly, the result was not completely the same.
  - The hash history based approach was able to detect the equality of versions while the version vector reports it as a conflict.
- Hash history was able to capture the state when two sets of non-commutative operations produce the same result independently.
  - We also found that this property helps more to reduce the number of conflicts in overall system,
  - especially when the merge process was able produce the same version regardless of which site merged the conflicting versions.
  - It does not necessarily require the operations (writes) are commutative.
- We believe this is quite interesting
  - Since most applications for optimistic operations are semi-commutative,
  - Meaning that some operations (writes) are commutative some are not.
  - In other words, some schedules of operations would produce the same result although the operations are not always commutative.
  - We believe this has been verified by the fact that many conflicting operations are automatically resolvable per application-specific semantics.