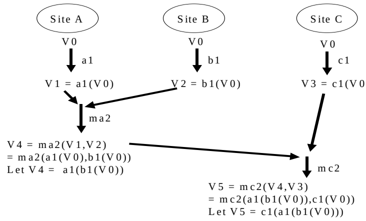
digital library project
University of California, Berkeley

Hash-History Approach for Reconciling Mutual Inconsistency in Optimistic Replication

B. Hoon Kang, Robert Wilensky and John Kubiatowicz ({hoon,wilensky,kubi}@cs.berkeley.edu)
CS Division, UC Berkeley

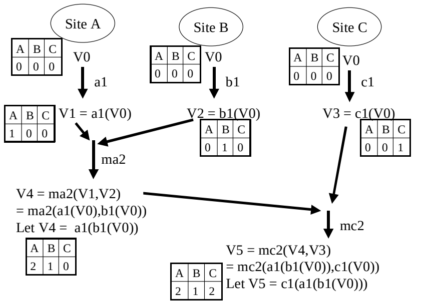
Optimistic Replication

- 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.



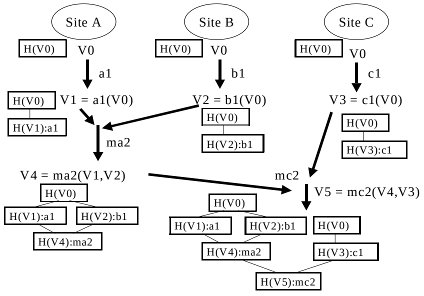
- Version Graph Example Above,
  - where a1, b1, and c1 are operations (or delta). ma2 and mc2 are merge operations.
  - Each site originally has V0.
- Site A initiates reconciliation with site B and site A merges the state (i.e version) V1 and V2 by deciding the ordering as b1 and a1.
  - If site B merges V1 and V2 the ordering may be different than that of site A.
  - 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.

Previous Approaches: Version Vectors



- Version Vectors are widely used in reconciling replicas
  - In weakly consistent replication systems(Bayou and Ficus)
  - Not assuming a synchronized timestamp nor centralized update serializer
- Doesn’t scale as number of replicas increases
  - Version vector needs one entry for each replica
  - Size of vector grows in proportion to number of replicas
  - Complexity of management grows as new replicas added or deleted

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 ancestral version can be found, if no version dominates
    - Useful hints in a subsequent diffing/merging
- Scalable to thousands of sites
  - Hash lists grows 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

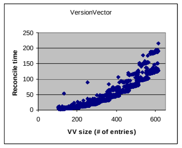
Hash History Pruning

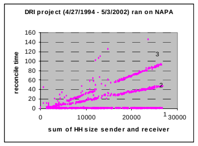
- Aging with Loosely Synchronized Clocks
  - The classical techniques
    - The global-cutoff timestamp (Lynch et al) and the acknowledgement-timestamp (Golding et al)
    - however, this method fundamentally requires to track the committed state per each site,
    - hence it would not scale to thousands of sites.
  - We chose to use the simplistic aging method based on roughly synchronized timestamp.
- Highly Sharable 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.
- *Pruning with CSN and OSN*
  - CSN (a monotonically increasing commit sequence number assigned by the primary site) to determine the fact a certain version belongs to the retired section or not.
  - A primary site can declare a retired version using OSN (Omitted Sequence Number)
  - Each site can prune its hash history aggressively by recording the OSN (Omitted Sequence Number) as the largest CSN of all the retired writes.
  - Later, the OSN is compared with the CSN of the latest version from the other site.
    - If OSN is bigger (newer) than CSN then the latest version from the other site is too old to be considered.

Simulation Result

	Freenet	Dri	Squirrel-mail
Events	2281	10137	1077
Users	64	21	39
Duration	12/28/1999-4/25/2002	4/27/1994 - 5/3/2002	11/18/1999 - 4/17/2002

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


VersionVector


DRI project (4/27/1994 - 5/3/2002) ran on NAPA

- We measured the reconcile time that needs to be spent
  - to determine the version dominance and
  - to manage the data structures
- The reconcile time
  - for hash history grows linearly as a function of number of revisions (writes) and
  - for the version vector as a function of number replica sites.
- The reconcile time as a function of HH size.
  - Line 1 indicates the constant time table lookup is happened and
  - line 2 indicates the time to accept the hash history when one dominates the other.
  - Line 3 shows the time to do conflict merging which involves updating both sender and receiver hash history

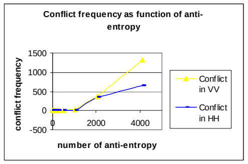
Correctness Test

	Total	A==B	A dom B B dom A	Conflict
Freenet VV	1315	20	291	1004
Freenet HH	1315	20	291	1004
Smail VV	1556	44	532	980
Smail HH	1556	44	532	980
Dri VV	6629	543	3289	2797
Dri HH	6629	543	3289	2797

- 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 deltas produce the same result.
  - Since version vector pessimistically assume such case, the results was exactly the same as in the table above.

Effectiveness Analysis

	Total	A==B	AdomB BdomA	Conflict
Freenet VV	1315	29	311	975
Freenet HH	1315	36	411	868
Smail VV	1556	39	480	1037
Smail HH	1556	48	564	944
Dri VV	6629	513	3395	2721
Dri HH	6629	907	3989	1733


Conflict frequency as function of anti-entropy

- We use a deterministic merge policy to simulate a deterministic merge behavior
  - so that two different schedules of deltas provide ***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 incremented 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 writes.
  - It does not necessarily require the operations (writes) are commutative.
- We believe this is quite interesting
  - since most applications for optimistic replications are semi-commutative,
    - meaning that some operations (writes) are commutative some are not.
    - In other words, some schedule 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

OceanStore Summer Retreat, June 10-12, Lake Tahoe