Conflict resolution in Kai

Shun'ichi Shinohara (shino)

@ Erlang/distributed workshop #2
2008-11-20 (Thu), KLab, Tokyo

Goals

- Learn that events are NOT total ordered, but can be "concurrent" in a distributed system.

- Understand some points on conflict resolution in Amazon's Dynamo.

- Discuss on Kai's designs and implementations concerning conflict resolution.

Contents (1/2)

I. Lamport's logical clock

- "happens-before" and "concurrent".
- Construction and features.
- Gleaning.

II. Vector Clocks

- Construction.
- keeping slender (practice).

Contents (2/2)

III. Some points on Amazon's Dynamo

- Coordinators increment vector clocks.
- Client-side conflict resolution.
- Multiple versions in a single node.

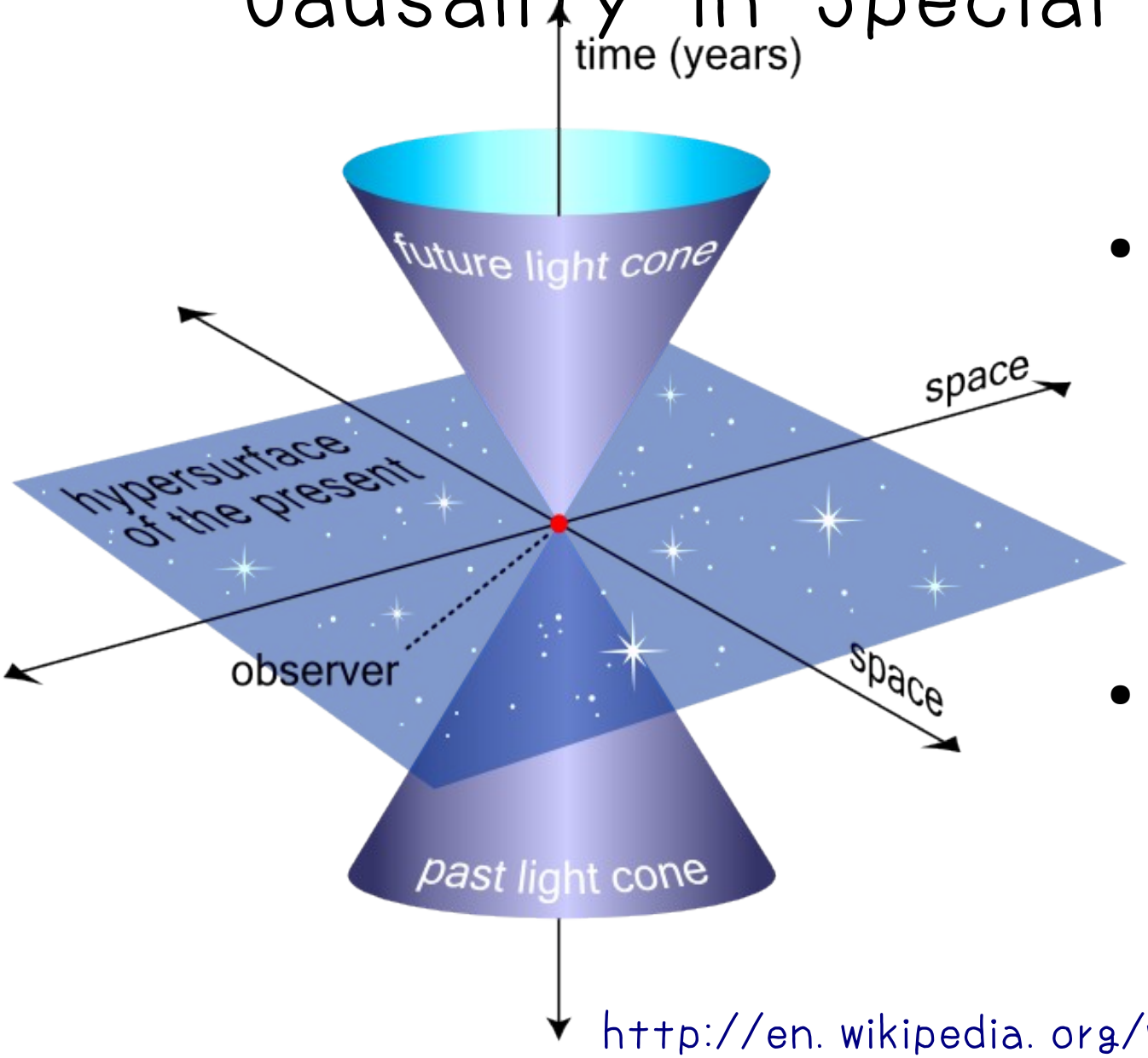
IV. Discussions on Kai's conflict resolution

- Packing checksums to "cas_unique" (lossy).
- How/Whether to merge a set of vector clocks.
- Consistency in putting data.

I. Lamport's logical clock

- Idea
  - Two events are NOT ALWAYS ordered in a distributed system.
  - Orders for causally-related events are only meaningful.
  - Inspired by A. Einstein's special relativity.

Causality in Special Relativity

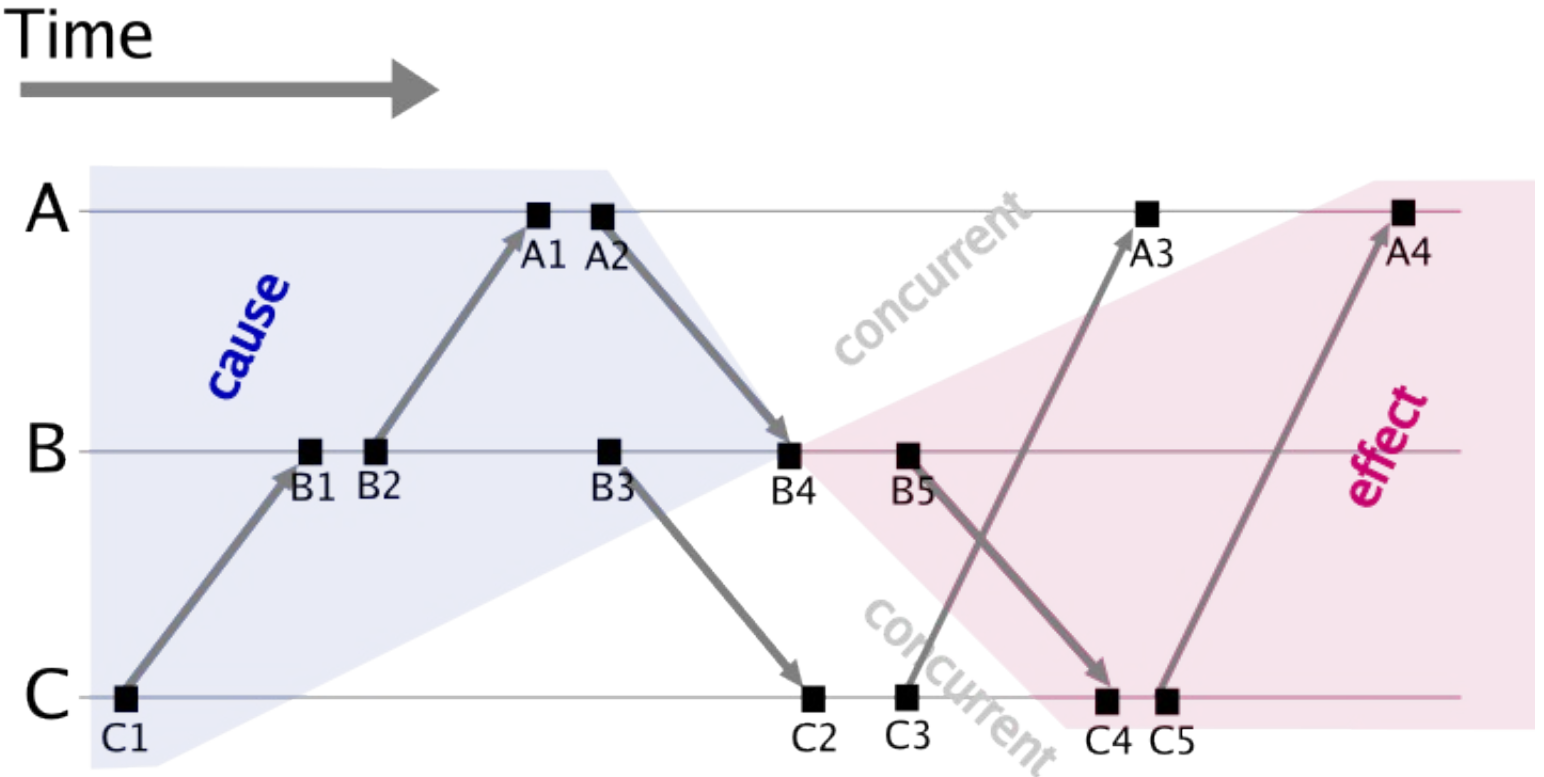


- Three types of regions
  - Past,
  - Future,
  - Simultaneous.
- Simultaneous is
  - not a plane,
  - but a region.

http://en.wikipedia.org/wiki/Light_cone

Problem setting

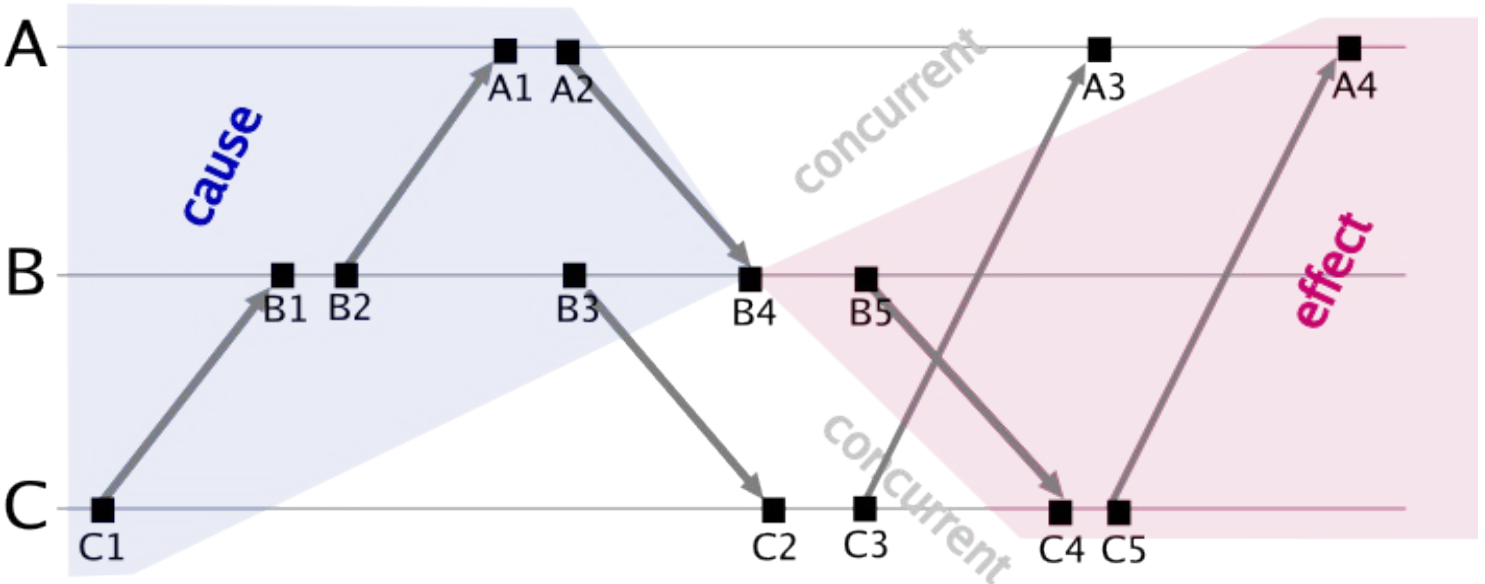
- Processes communicate by passing messages.
- Messages <==> causal relationships.



http://commons.wikimedia.org/wiki/Image:Lamport-Clock-en.svg

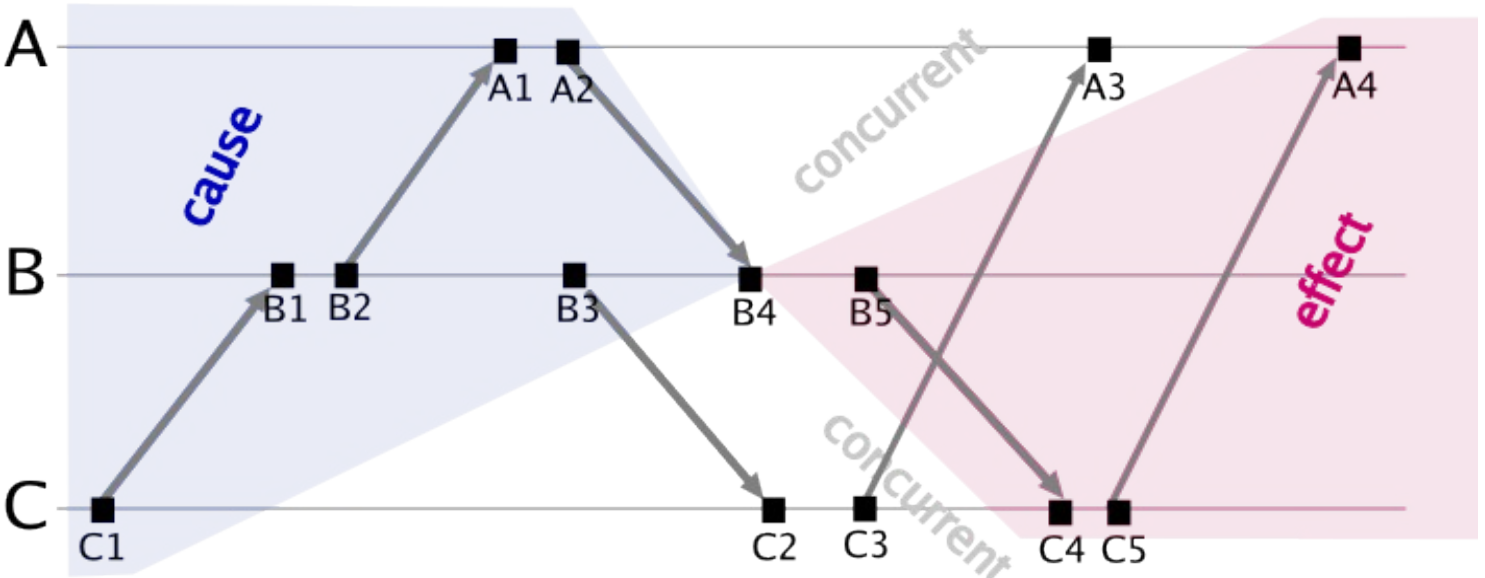
“happens-before” (->)

- In a single process, -> means the “normal” order, e.g. A1 -> A2.
- E1/E2 are sending/receipt of the same message, then “E1 -> E2”, e.g. A2 -> B4.
- “E1 -> E2 and E2 -> E3” then “E1 -> E3”, e.g. A1 → B4.



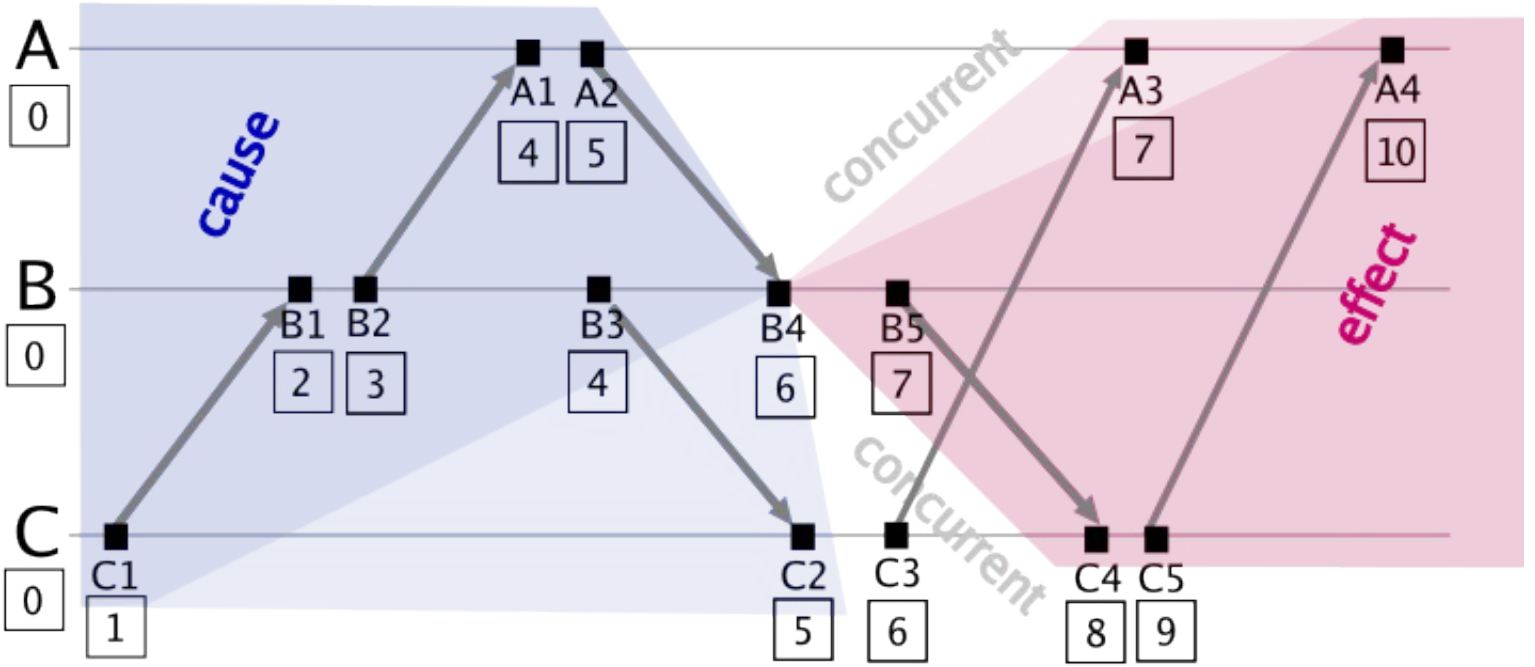
“concurrent” (||)

- “not (A -> B)” and “not (B -> A)”.
- e. g.　A2 and B3,　B4 and C2.



Construction

- Increment when any event occurs, and
- Message receivers increment their clock to
  - max(present clock, sender's clock) + 1.



http://commons.wikimedia.org/wiki/Image:Lamport-Clock-en.svg

Features

- When A -> B, C(A) < C(B). ---(*)
  - Clock order is consistent with causality.
- But: C(A) < C(B) does not imply A -> B.
  - Clock order does NOT determine a unique causality-relation.
- Just: C(A) <= C(B) implies "not (B -> A)"
  - The contraposition of (*).
  - "not (B -> A)" is equivalent "A -> B or A || B".

Gleaning

- Extension to total ordering.
  - Label an event as (process id, logical clock).
  - An (arbitrary) total order in {process id}.
  - (P(A), C(A)) < (P(B), C(B)) if and only if
    - C(A) < C(B), or
    - C(A) = C(B) and P(A) < P(B).
- Example: mutex algorithm
  - Non centralized, but all nodes should participate.
  - Consult the paper for details :-)

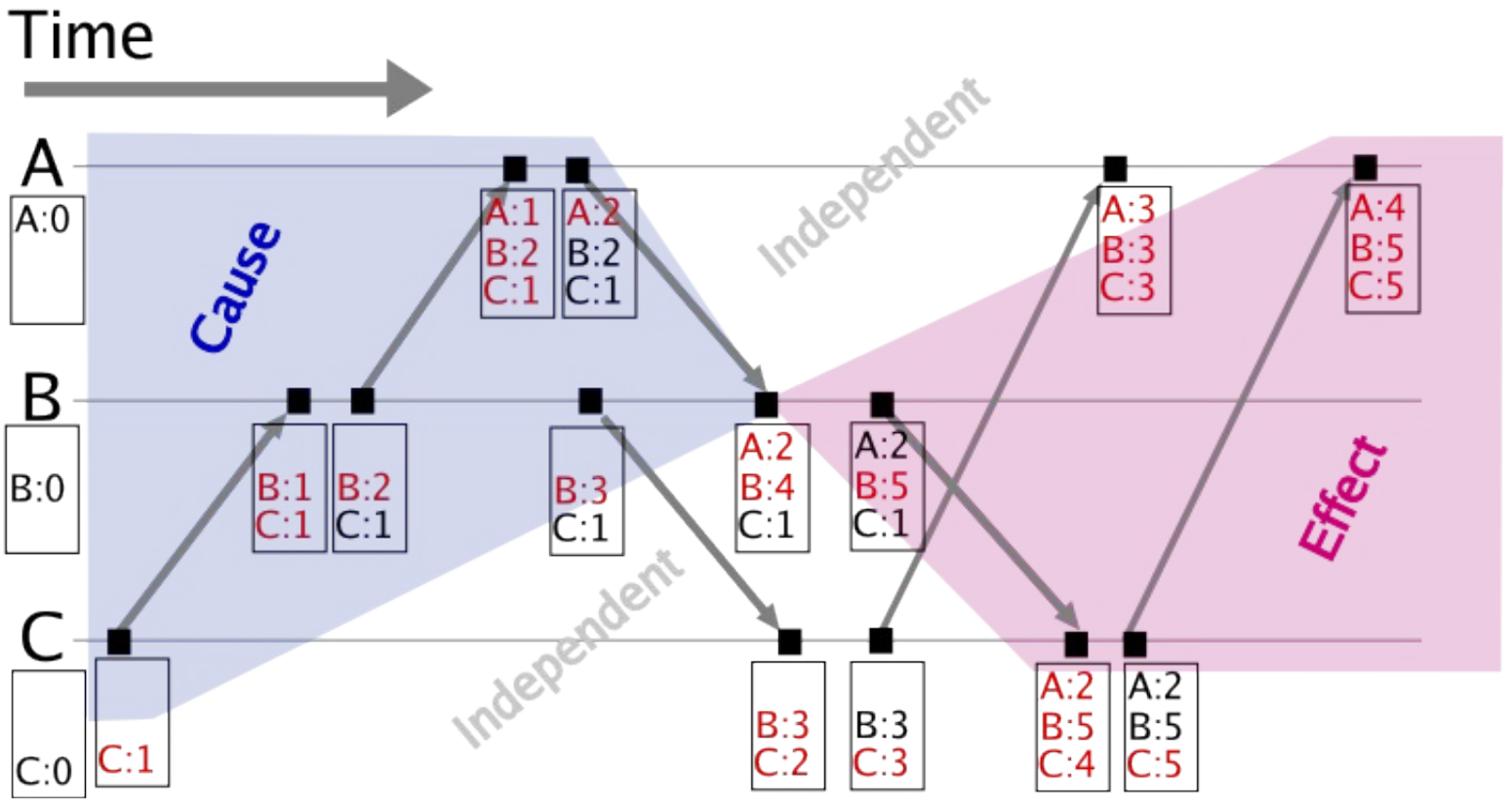
II. Vector Clocks

- Vector clocks provides the clock s.t.
  - VC(A) < VC(B) then A -> B,
  - VC(A) || VC(B) then A || B,
  - Where VC(A) is the vector clock of an event A.

- It determines causality relation, not only "happens-before", but also "concurrent" for events.
- In practice: keeping them slender.

Construction

- A vector clock: a set of {process id, clock}.
- Increment own clock and merge (as follows).



http://commons.wikimedia.org/wiki/Image:Vector_Clock.svg

Keeping slender (practice)

- Element size can be bloated.
- As a size = #(related nodes) grows,
  - Comparison slows down: not so bad (usually),
  - Data consumes space (memory/storage): bad,
  - Messages become fat: TOO BAD!

- Related nodes should be as few as possible.

III. Some points from Amazon's Dynamo

- ☞ Coordinators increment vector clocks.
- ☞ Client-side conflict resolution.
- ☞ Multiple versions in a single node.

- This part includes my guess.

Coordinators increment vector clocks

- In section 4.3 in the Dynamo paper.
  - 4.3. Replication
    - "*Each key, k, is assigned to a coordinator node [...]. The coordinator is in charge of the replication of the data items that fall within its range.*"
- Without failures, every vector clock has just one element.
- Suppress vector clocks to be fat.

Client-side conflict resolution

- If concurrent versions exist in server-side,
  - A Client receives multiple data, each with vector clocks,
  - The client merge
    - Not only data (e.g. for hash-type data, just "merge" them),
    - But vector clocks.
- Servers don't know data schema.
- Both servers and clients know vector clock manipulation.

Multiple versions in a single node

- In section 6 and 6.4
  - 6. Experiences & Lessons Learned
    - *"All the measurements presented in this section were taken on a live system operating with a configuration of (3, 2, 2)"*
  - 6.3 Divergent Versions: When and How Many?
    - *the number of versions returned [..]*
      99.94% of requests saw exactly one version; [..]
      and 0.00009% of requests saw 4 versions.
- A single node stores multiple conflicting data possibly.

IV. Discussions on Kai's implementation

- Packing checksums to "cas_unique"
- How/Whether to merge a set of vector clocks
- Consistency in putting data

Packing checksums to “cas_unique”

- Kai uses memcached protocol between clients and servers
- Memcached’s “gets” and “cas” commands uses “cas_unique” field (uint64) for optimistic lock
- Mapping of Dynamo operations to memcached command:
  - GET => gets,
  - PUT => cas.

Packing checksums to "cas_unique"

- Packing one checksum (no concurrent data).
  - <<1:4, [data's checksum]:60>>
- Packing two checksums.
  - <<2:4, [data1's checksum]:30,
    [data2's checksum]:30>>
- General cases are
  - First 4 bit: #(data),
  - [Each data's checksum]:(60/#(data)),
  - Zero padding rest.
- Can ONLY check consistency (lossy).

How/Whether to merge a set of vector clocks

- Clients don't merge vector clocks in Kai
- Who merges them???
- kai_coordinator can merge the clocks in cas,
  - At the cost of some message delay.
    - Receive "cas" command,
    - Collect data from replica nodes (just as "get"),
    - Check consistency, cas_unique <==> vector clocks,
    - Merge vector clocks,
    - Store data in each replica node.
  - Spoils the advantage of (N, R, W) flexibility.

Alternatives (1/2)

- Break the memcached protocol in
  - *"The server will transmit back unstructured data in exactly the same way it received it" (from the memcached protocol).*
- Sketch of implementation:
  - Merge related vector clocks,
  - Serialize the merged vector clock,
  - Use cas_unique to represents its length
  - Concatenate the serialized vector clock and the user data and send in a data block.

Alternatives (2/2)

- Server side resolution,
  - e.g. a user can add-on a module of resolution logic.
- Other protocols,
  - e.g. HTTP, XMPP(i don't know much).
  - Memcached's client libraries by C and various language wrappers are appealing.
- Nice idea?

Consistency in putting data

- Problem
  - Operations go on multiple nodes.
  - How to guarantee the consistency between updating vector clocks and storing data?

- "Multiple data in a single node" can help?
  - Appropriate to Dynamo's "always writable" policy.

Summary

- ☞ Causality in a distributed system has only partial order.
- ☞ Vector clocks is easy to implement/use, but should pay attention to keep them slender.
- ☞ "Multiple versions in single node" seems nice.

- Please discuss anything later and at MLs.

References

- Lamport's logical clock
  - http://research.microsoft.com/users/lamport/pubs/pubs.html#time-clocks
  - http://www.cs.tsukuba.ac.jp/~yas/sie/csys-2007/2008-02-29/
- Vector clocks
  - http://www.vs.inf.ethz.ch/publ/papers/VirtTimeGlobStates.pdf
  - http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.7435
  - http://www.slideshare.net/takemaru/kai-an-open-source-implementation-of-amazons-dynamo-472179
- Dynamo
  - http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html
- Kai
  - http://kai.wiki.sourceforge.net/
- Memcached protocol
  - http://code.sixapart.com/svn/memcached/trunk/server/doc/protocol.txt