Causal Consistency for Distributed Data Stores and Applications as They are

Kazuyuki Shudo, Takashi Yaguchi
Tokyo Tech
Background: Distributed data store

- Database management system (DBMS) that consists of **multiple servers**.
  - For performance, capacity, and fault tolerance
  - Cf. **NoSQL**

- A data item is **replicated**.

![Diagram of a distributed data store with replicas and servers](chart.png)

A **cluster of 1 - 1,000 Servers**

- **NoSQL**:
  - Apache Hbase
  - Cassandra
  - Riak

The base of our implementation
Background: Causal consistency

• One of consistency models.

• A consistency model is a contract between DBMS and a client
  – of what a client observes.
  – It is related to replicas closely. If a client see an old replica, ...

• Consistency models related to this research:
  – Eventual consistency
    • All replicas converge to the same value eventually.
    • Most NoSQLs adopt this model.
  – Causal consistency
    • All writes and reads of replicas obey causality relationships
      between them.
Background: Causal consistency

- **An example:** social networking site

  - **Causally consistent**
    - Now I’m in Atlanta!
    - It’s warmer than I expected.

  - **Not causally consistent**
    - A client does not see the original post

- **Precise definition**
  - **Write after read** by the same process (client)
  - **Write after write** by the same process - illustrated above
  - **Read after write** of the same variable (data item) regardless of which process reads or writes
Contribution: Letting-It-Be protocol

- A protocol to achieve causal consistency on an eventually consistent data store.
- It requires no modification of applications and data stores.

**Data store approach**

Ex. COPS, Eiger, ChainReaction and Orbe

- Applications
  - Access
  - Eventually consistent data store

**Middleware approach**

**Existing protocol**

Ex. Bolt-on causal consistency

- Applications
  - Modified to specify explicitly data dependency to be managed
  - Middleware
    - Eventually consistent data store

**Our Letting-It-Be protocol**

does not require any modifications to either data stores or applications

- Applications
  - Middleware
    - Eventually consistent data store

------ Modified part of software
Causality resolution in general

• Servers maintain dependency graphs and resolve dependency for each operation.
Causality resolution

• **Data store approach** – **write time**
  - When a server receives a replica update of \( v3 \), before writing \( v3 \), the server confirms the cluster has level 1 vertexes, \( x1, y2 \) and \( z1 \).
    - \( u4 \) is confirmed when \( z1 \) is written.

• **Middleware approach** – **read time**
  - It cannot implement write-time resolution.
    - Because a middleware cannot catch a replica update.
  - When a server receives a read request of \( v \), the server confirms that the cluster has all the vertexes including \( x1, y2, z1 \) and \( u4 \).

![Dependency graph for v3](image)

- \( U4 \), Level 2
- \( X1, Y2, Z1 \), Level 1
- \( V3 \), Level 0
Problems of middleware approach

It requires no modification of a data store. But there are problems.

• Overwritten dependency graph
  – Dependency graph for \( v4 \) overwrites graph for \( v3 \) though it is still required as part of graphs for other variables.
  – Solution: … (in the next page)

\[ \text{Dep graph for } v \quad \text{Dep graph for } t \]

\[ V_3 \] is to be overwritten by \( v4 \).

\[ t \]

\[ \text{can be lost.} \]

• Concurrent overwrites by multiple clients
  – Multiple \( v3 \) are written concurrently.
  – Solution: Mutual exclusion with CAS and vector clocks.
Solutions to overwritten dependency graph problem

• **Bolt-on** attaches entire graph (!) to all the variables.
  – It reduces the amount of data by forcing an app to specify deps explicitly.
  – It requires *modification of apps*. 😞

• Our **Letting-It-Be** keeps graphs for multiple versions such as $v4, v3$.
  – It reduces the amount of data by attaching only level 1 vertexes.
  – It requires *no modification of apps*. 😊
  – It traverses a graph across servers 😞, but marking technique reduces it.
  – It requires garbage collection of unnecessary old dep graphs. 😞
Performance

- Our contribution is a protocol that requires no modification of both apps and a data store.
- But, performance overheads should be acceptable. It depends on an application.

Benchmark conditions
- 2 clusters, each has 9 servers running Linux 3.2.0, and 50 ms of latency between the clusters
- Apache Cassandra 2.1.0, configured as each cluster has one replica.
- Letting-It-Be protocol implemented as a library in 3,000 lines of code
- Yahoo! Cloud Serving Benchmark (YCSB) [ACM SOCC 2010] with Zipfian distribution
Performance

**Best case:** Read latencies with read-heavy workload

**Worst case:** Write latencies with write-heavy workload

- Overheads for reads are smaller than writes though the protocol does read-time resolution.
  - Marking already-resolved data items works well.
- Comparison with Bolt-on is part of future work.
Summary

• Letting-It-Be protocol maintains causal consistency over an eventually consistent data store.
  – We demonstrated that it works with a production-level data store, Apache Cassandra.

• It is unique in that it requires no modifications of applications and a data store.

• Future direction
  – A better consistency model that involves
    • less modification to each layer,
    • less costs,
    • less and simple interaction between layers,
    • easier extraction of consistency relationships from an application.