Don’t settle for Eventual Scalable Causal Consistency for Wide-Area Storage with COPS

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen
Princeton University, Intel Labs, Carnegie Mellon University

Presentation by Ankit Shah and Barrett Olson
Background
Background: ALPS

- Availability
- Low Latency
- Partition tolerance
- Scalability

Current set of desirable qualities for real world distributed systems.

Trade-offs: provably not compatible with strong consistency (CAP theorem)!
Review: Consistency Models

- **Linearizability**—sometimes informally called strong consistency
  - Global order with real time limitations
  - E.g. Paxos
- **Sequential** consistency
  - Some valid global order
- Causal consistency
- Eventual consistency
  - We’ll get there... eventually
  - E.g. Dynamo

**Linearizability** and **Sequential consistency** not compatible with ALPS (aforementioned CAP theorem)
Review: Causal Consistency

- **Execution Thread**
  - If a and b are two operations in a single thread of execution, then a --> b if operation a happens before operation b

- **Gets From**
  - If a is a put operation and b is a get operation that returns the value written by a, then a --> b

- **Transitivity**
  - For operations a, b, and c, if a --> b and b --> c, then a --> c
Review: Causal Consistency

Client 1: put(x, 1) → put(y, 2) → put(x, 3)

Client 2: get(y) = 2 → put(x, 4)

Client 3: get(x) = 4 → put(z, 5)

Time: ____________________________,→
Issue with causal

- Case where a \(\rightarrow\) b and b \(\rightarrow\) a
  - Also known as concurrency
- Causal consistency allows for conflicts
  - Permanent divergence becomes valid

\[
\begin{align*}
\text{Replica 1} & \quad \text{Replica 2} \\
\text{put(x, 3)} & \quad \text{put(x, 4)} \\
\text{put(x, 4)} & \quad \text{put(x, 3)} \\
\text{get(x) = 4!} & \quad \text{get(x) = 3!}
\end{align*}
\]
Introducing Causal+ consistency

- Causal consistency with convergent conflict handling
  - Solves problem proposed on previous slide
- Stronger than Causal consistency, but still can be achieved in ALPS systems!
Convergent conflict handling

- Conflicting writes must be handled in the same way at all replicas
- Abstraction: handler function
  - Commutative and Associative function that allows for replicas to handle writes in order of receiving
- Common way to do so - last writer wins
- Can also be more complicated
  - E.g. send both values of write to client and have it decide
Causal+ examples

- Causal consistency
  - Uploading and viewing a photo album
- Convergent Conflict handling
  - Multiple people updating event times
Maintaining Causal+ in COPS

- Versions on keys (different version per value)
  - Once version $i$ of a key is returned, only that version or more recent may be returned
  - Conflict resolution is considered causally after writes

- Dependencies
  - Maintained to ensure that past writes are completed before doing current
COPS System
COPS System design

- Key-Value Store
- Provides: ALPS Properties & Causal+ consistency
- As noted, uses Versions & Dependencies
  - Dependencies are the reverse of the causal ordering
- Asynchronous Replication
  - Between COPS clusters in different datacenters
Two Versions of COPS

- **COPS**
  - Data store that is causal+ consistent
  - Lower overhead

- **COPS-GT**
  - Superset of COPS
  - Support for 'get transactions'
  - Needed Additional Metadata (full dependency list)
The COPS Architecture (Fig. 4 in the paper)
Client Library and Interface
**Client Interface**

- Uses the COPS client library
  - exposes a put/get interface
  - ensures operations are properly labeled with causal dependencies
- Application is running in the same datacenter
- Clients communicate only with their local COPS cluster
Client API

- `ctx id ← createContext()`
- `bool ← deleteContext(ctx id)`
- `bool ← put (key, value, ctx id)`
- `value ← get (key, ctx id) [In COPS]` OR
- `<values> ← get_trans (<keys>, ctx id) [In COPS-GT]`
Add Metadata to API Call

- Client library adds dependency metadata
- puts are sent to clusters as:
  - \langle \text{bool, vers} \rangle \leftarrow \text{put\_after (key, val, [deps], nearest, vers=∅)}
- gets are sent to clusters as:
  - \langle \text{value, version, deps} \rangle \leftarrow \text{get\_by\_version (key, version=LATEST)}
put example (client)

Client 1

put(Key, Val)

deps

...K_version

put_after(Key, Val, deps)

version

(Thread-Of-Execution Rule)

Credit: Wyatt Lloyd, cops-sosp11-talk-public
get example (client)

Client 2

get(K)

get(K)

value

deps

...K

version

L_{337}

M_{195}

get(K)

value, version, deps'

(Gets-From Rule)

(Transitivity Rule)

deps'

L_{337}

M_{195}

Credit: Wyatt Lloyd, cops-sosp11-talk-public
Local COPS Cluster

Data Store Node

- Key1 = [<1,V,D>,<2,V,D>]
- Key2 = [<2,V,D>]
- Key3 = [<1,V,D>,<3,V,D>]

Repl Queue

<table>
<thead>
<tr>
<th>Key : Vers</th>
<th>Value</th>
<th>Deps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Put_after

Get_by_vers

dep_check
Local COPS Cluster

- Locally linearizable key-value store
- Partitioning the keyspace into N partitions
  - Keys are assigned primary node
  - Replicated across local nodes using chain replication
- Complete replica of the stored data
  - Simplifies system
  - Other options possible (not explored in this paper)
    - But would sacrifice low latency
put example (local cluster)

- Wait for dep_checks return
  - then expose value
- Ensures Causal+

Credit: Wyatt Lloyd, cops-sosp11-talk-public
Geo-replicated COPS Clusters
Wide-Area Replication

- After a write commits locally
- Places write in a replication queue
- Asynchronously replicates that write to different clusters
  - using a stream of put_after operations
- Each cluster will use dep_check calls to keep Causal+
  - Same as in local cluster
put example (geo-rep cluster)

- Each cluster is sent a put_after request
- Clusters will use dep_check on the put_after's deps
- Making sure all deps are satisfied locally
  - before committing the write

Credit: Wyatt Lloyd, cops-sosp11-talk-public
Get Transaction Algorithm

- Algorithm in two rounds
- First Round
  - For each key the client listed in get_trans:
  - Issue concurrent get_by_version operations to the local cluster
- Second Round
  - For all keys/dependencies not satisfied that were returned in the first round:
  - Issue a second round of concurrent get_by_version operations
    - Specifying an explicit version: avoids adding new dependencies
Does this dependency tracking sound familiar?

- At first glance, COPS is basically
- IMPLEMENTING STRONG CLOCKS
- THE HARD WAY!!!
- Gross...
IMPLEMENTING STRONG CLOCKS
(the hard way)

\[
\begin{align*}
\{a\} & \quad \{a, b\} & \quad \{a, b, c\} & \quad \{a, b, c, d, e, f\} \\
\bullet \quad a & & b & & c & & d \\
\{a, b, e\} & \quad \{a, b, e, f\} & \quad \{a, b, e, f, h, g\} \\
\bullet & & e & & f & & g \\
\{h\} & \quad \{h, i\} & \quad \{a, b, c, h, i, j\} \\
\bullet & & h & & i & & j
\end{align*}
\]

Strong clock condition: \( p \rightarrow q \iff \theta(p) \subset \theta(q) \)
Optimizations
Optimization: Nearest Dependency

- We only need to check the nearest dependencies (for individual keys).
- E.g. $z_4$ dependencies are satisfied if $y_1$ and $v_6$ are committed locally.
- However, the full dependency list is needed for `get_trans` operations in COPS-GT.
- More optimizations are needed to make the system practical.

<table>
<thead>
<tr>
<th>Val</th>
<th>Nearest Deps</th>
<th>All Deps</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_2$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$u_1$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$v_6$</td>
<td>$t_2, u_1$</td>
<td>$t_2, u_1$</td>
</tr>
<tr>
<td>$w_1$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$x_3$</td>
<td>$w_1$</td>
<td>$w_1$</td>
</tr>
<tr>
<td>$y_1$</td>
<td>$x_3$</td>
<td>$x_3, w_1$</td>
</tr>
<tr>
<td>$z_4$</td>
<td>$y_1, v_6$</td>
<td>$t_2, u_1, v_6, w_1, x_3, y_1$</td>
</tr>
</tbody>
</table>
Optimization: Garbage Collection

- **Version Garbage Collection (for COPS-GT)**
  - Limit the total running time of `get_trans -> 'trans_time'`
  - No need to store old versions longer than `trans_time` (plus clock skew)

- **Dependency Garbage Collection**
  - Using 'trans_time' and Causal+ consistency’s progressing property
  - Remove dependencies once committed to all replicas (see example in paper)

- **Client Metadata Garbage Collection**
  - never-depend flag or global checkpoint time
Fault Tolerance

- Primarily just use existing techniques (No BFT at all)
- Client Failure -> no recovery necessary
- Key-Value Node Failures -> Chain replication*
- Datacenter Failures -> ... (not great)
  - put_after ops from failed datacenter, not copied out, will be lost
  - storage required for replication queues in the active datacenters will grow
  - COPS will be unable to garbage collect dependencies...
    - system administrator intervention?
Conflict Handling (the '+' of causal+)

- Two “simultaneous” writes to a given key -> Conflict
- put operations have prev-version as metadata dependency
- Lamport timestamp is used to assign a unique version number to each update
- timestamp plus node Id gives use a global order over all writes for each key
- Simple last-writer-wins policy
Evaluation
Methodology

- Built COPS and COPS-GT on top of a key value store
- Partitioned
- Client sends numerous put/get requests
  - System becomes saturated
  - Sampling to decide what key gets hit

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Parameter</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>datacenters</td>
<td>2</td>
<td>put: get ratio</td>
<td>1:1 or 4</td>
</tr>
<tr>
<td>servers / datacenter</td>
<td>4</td>
<td>variance</td>
<td>1</td>
</tr>
<tr>
<td>clients / server</td>
<td>1024</td>
<td>value size</td>
<td>1B</td>
</tr>
<tr>
<td>keys / keygroup</td>
<td>512</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Probability plot](image-url)
Results

Figure 9: Maximum throughput and the resulting average dependency size of COPS and COPS-GT for a given inter-op delay between consecutive operations by the same logical client. The legend gives the put:get ratio (i.e., 1:9 or 1:4).

Figure 10: Maximum throughput and the resulting average dependency size of COPS and COPS-GT for a given put: get ratio. The legend gives the variance (i.e., 0, 1, or 512).
Conclusion

- Causal+ consistency can be achieved within ALPS systems
  - Useful for constant interaction applications
- Future work
  - Where is COPS?
  - Read-only optimizations - SNOW
Some Common Questions

- **Question:** Why not just use vector clocks?
  - Author's Answer: They are not compatible with "distributed verification"

- **Question:** How do you deal with different types of failures?
  - Author's Answer: "we just used existing techniques to deal with failures"

- Personally we find the advantage not using vector clocks provide to be a little dubious
  - What do y'all think?
Thank you!
Questions? Comments? Concerns?