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

Authors: Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, David G. Andersen
Presenter: Kailin Tang
Outline

• Background & Motivation of COPS

• System Design of COPS

• Experiments & Evaluation

• Conclusion
Motivation

• Large number of online applications
  • Search engine
  • Social network
  • Online Shopping

• Problem with single data center in operation
  • Latency?
  • Availability?

• Solution:
  • Geo-replicated data centers
Motivation – Geo-replicated Distributed Data Store
Two kinds of replication involved:

- Replication within a DC
- Replication between DCs
Motivation – Desired Properties (ALPS)

• **Availability**
  • Operations always complete successfully

• **Low latency**
  • Operations finish quickly

• **Partition Tolerance**
  • System continues to operate even if there is a separation in network

• **Scalability**
  • Easy to increase capacity and throughput of storage tier in each DC

“Always On”
Motivation – Desired Properties: Consistency

• Restricts ordering/timing of operations

• Stronger consistency
  • Makes programming easier
  • Makes user experience better

• However......
Fact: It is impossible to achieve ALPS and strong consistency at the same time.
Motivation – Which Consistency Model (with ALPS)?

- **Strong (Linearizability)**
  - Impossible [Brewer00, GilbertLynch02]

- **Sequential**
  - Impossible [LiptonSandberg88, AttiyaWelch94]

- **Causal**
  - COPS

- **Eventual**
  - Amazon Dynamo
  - LinkedIn Voldemort
  - Apache Cassandra

**Idea:**
Pick the strongest consistency model which is compatible with ALPS.
Quick Review – Potential Causality

• Three Rules:
  • 1. **Execution Thread.** If $a$ and $b$ are two operations in a **single thread of execution**, then $a \rightarrow b$ if operation $a$ happens before operation $b$.

  • 2. **Gets From.** If $a$ is a **put operation** and $b$ is a **get operation** that returns the value written by $a$, then $a \rightarrow b$.

  • 3. **Transitivity.** For operation $a$, $b$ and $c$, if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. 
Quick Review – Potential Causality Example

• Causal Relationship Graph

Causal consistency requires that values returned from get operations at a replica are consistent with the order defined by causality (happen before).

Causal consistency does not order concurrent operations.
Motivation – Why Causal Consistency?

- Remove your boss from your friend list
- Post to friends: “Time for a new job!”
- Friends read post
- You’re fired!

- Upload a photo
- Add photo to album
- Friend view the photo
Motivation – Causality is Useful!

For Users:
- Friends
- New Job!

For Programmers:
- Photo Upload
- Add to album

Employment Integrity
Referential Integrity
Motivation – Conflict Problem

• Concurrency
  • Two people in two different places write to the same key at the same time
Motivation – Conflict Problem & Solutions

• Conflicts are undesirable 😞
  • Conflicts allow replicas to diverge forever
    • K = 5 in DC1 while K = 7 in DC2
  • Conflicts may represent an exceptional condition
    • Online shopping cart example

• Convergent conflict handling
  • Last-writer-wins
    • Simple but limited
  • Application specific conflict handler functions
    • Flexible but complicated
Motivation – A New Consistency Model

**Causal Consistency**

Causal component ensures that the data store guarantees the causal dependencies between operations

**Convergent Conflict Handling**

Convergent conflict handling component ensures that replicas never permanently diverge and that conflicting updates to the same key are dealt with identically at all sites

**Causal+ Consistency**

Clients see only progressively newer versions of keys
What makes COPS innovative?

• Previous Causal+ Systems
  • Bayou ‘94, TACT ‘00, PRACTI ‘06
  • Log-exchange based
    • Write incoming operation to log
    • Send logs to other replicas
    • Other replicas replay operations in the log
  • Log is single serialization point that implicitly capture and enforce casual order
  • Problem: Every operation in a DC need to be serialize to one point result in poor scalability

• Key idea in COPS to achieve high scalability
  • Dependency metadata explicitly captures causality
  • Distributed verifications replace single serialization
    • Delay exposing replicated puts until all dependencies are satisfied in the DC
Outline

• Background & Motivation of COPS

• System Design of COPS

• Experiments & Evaluation

• Conclusion
What is COPS?

COPS (Clusters of Order-Preserving Servers) System is a Key-Value storage system provides causal+ consistency and is designed to support complex online applications that are hosted from a small number of large-scale datacenters across the wide-area.

• Two versions
  • COPS
  • COPS-GT
    • COPS + Get Transaction (consistent view of multiple keys)

• Build blocks
  • Frontend servers (clients of COPS)
  • Backend key-value stores

• Replications
  • Linearizability replication within DC with FAWN-KV
  • Causal+ consistency replication among DCs
COPS Architecture Overview

• System Components
  • Client library
    • Exposes put/get interface to client
    • Ensure operation are properly labeled with causal dependencies
  • Key-value store
    • Replicate data between clusters
    • Ensure writes are committed in their local cluster only after dependencies are satisfied
COPS Architecture – Client

• A client uses COPS client library for operations
• Client library maintains state about clients' dependency
  • context parameter
• APIs

\[
\text{bool} \leftarrow \text{put} \ (\text{key}, \text{value}, \text{ctx}_\text{id})
\]

\[
\text{value} \leftarrow \text{get} \ (\text{key}, \text{ctx}_\text{id}) \quad \text{[In COPS]}
\]

\[
\langle \text{values} \rangle \leftarrow \text{get}_\text{trans} \ (\langle \text{keys} \rangle, \text{ctx}_\text{id}) \quad \text{[In COPS-GT]}
\]

\[
\text{ctx}_\text{id} \leftarrow \text{createContext}()
\]

\[
\text{bool} \leftarrow \text{deleteContext}(\text{ctx}_\text{id})
\]
COPS Architecture – KV Store

- A standard key-value store provides linearizable operations on keys
  - Key space partitioned by consistent hashing for scalability
  - Each key is replicated across nodes by chain replication for fault tolerance

- Metadata
  - Version number
    - Most recent version (COPS)
    - List of versions (COPS-GT)
  - Dependencies
    - COPS-GT only
COPS Architecture – KV Store (cont.)

• APIs

\[
\langle \text{bool, vers} \rangle \leftarrow \text{put\_after}\ (\text{key, val, [deps], nearest, vers=0})
\]

\[
\text{bool} \leftarrow \text{dep\_check}\ (\text{key, version})
\]

\[
\langle \text{value, version, deps} \rangle \leftarrow \text{get\_by\_version}\ (\text{key, version=LATEST})
\]

• Keep old version of key-value pairs in COPS-GT
  • Support get transaction
  • Garbage collection required

• Asynchronous remote replication with replication queue
• Dependency checking mechanism ensure causally consistent order
Operations in COPS (COPS-GT) – Write

• Writes are first satisfied in local DC then replicated to remote DC
• Client calls the put client library

  \[ \text{bool} \leftarrow \text{put} \left( \text{key}, \text{value}, \text{ctx}_\text{id} \right) \]

• Library issues a write to the KV store

  \[ \langle \text{bool}, \text{vers} \rangle \leftarrow \text{put}_{-}\text{after} \left( \text{key}, \text{val}, \text{[deps]}, \text{nearest}, \text{vers}=\emptyset \right) \]

• Key’s primary storage node in local DC assigns the key a version number
  • Using Lamport timestamp to assign unique version number
• Library receive a response from KV store
  • Update client context metadata
• Return the value to the calling code
Operations in COPS (COPS-GT) – Write (cont.)

• Now we only update the value in local DC, we need to replicate this operation to remote DC
  • Causal+ Replication

• After a write commits locally, the primary storage asynchronously replicates write to remote DC by using put_after again

\[\langle \text{bool, vers} \rangle \leftarrow \text{put\_after}(\text{key, val, [deps], nearest, vers=0})\]
Operations in COPS (COPS-GT) – Write (cont.)

• When another cluster receive put_after, it checks whether the dependencies are satisfied locally by issuing dep_check within cluster

  \[ \text{bool} \leftarrow \text{dep\_check (key, version)} \]

• For the node receive dep_check, check dependencies
  • If satisfied, respond immediately
  • If not, block until need version has been written

• If all dep_check success, commit the write and expose value
  • Casual+ guarantee 😊
Operations in COPS – Read

• Reads are satisfied in local cluster
• Client calls the get library function

\[ value \leftarrow \text{get}(key, ctx_id) \] [In COPS]

• Library issues a read to the KV store

\[ \langle value, version, deps \rangle \leftarrow \text{get\_by\_version}(key, version=LATEST) \]

• Library receive a response from KV store
  • Add the \(<key, version, [deps]\)> to the client context
• Return the value to the calling code
The get operation in basic version of COPS only guarantee causal consistency for single key, it is not enough!
Operations in COPS-GT – Get Transaction

• Provide consistent view of multiple keys
  • Snapshot of visible values

• Keys can be spread across many servers
  • Provide consistent view spread across many servers in the system

• Take at most 2 parallel rounds of get
  
• No locks, no blocking

Low Latency 😊
Operations in COPS-GT – Get Transaction (cont.)

• Get transaction algorithm in at most 2 rounds

\[
\langle \text{values} \rangle \leftarrow \text{get\_trans}((\text{keys}), \text{ctx\_id}) \quad \text{[In COPS-GT]}
\]

```python
# @param keys list of keys
# @param ctx_id context id
# @return values list of values

function get_trans(keys, ctx_id):
    # Get keys in parallel (first round)
    for k in keys:
        results[k] = get_by_version(k, LATEST)

    # Calculate causally correct versions (ccv)
    for k in keys:
        ccv[k] = max(ccv[k], results[k].vers)
        for dep in results[k].deps:
            if dep.key in keys:
                ccv[dep.key] = max(ccv[dep.key], dep.vers)

    # Get needed ccvs in parallel (second round)
    for k in keys:
        if ccv[k] > results[k].vers:
            results[k] = get_by_version(k, ccv[k])

    # Update the metadata stored in the context
    update_context(results, ctx_id)

    # Return only the values to the client
    return extract_values(results)
```

- k concurrent get_by_version operation to local cluster and get k <value, version, deps> tuples
- Examine the dependency entries of every key and calculate causally correct versions
- Only if there are some keys which are not satisfied causality will the second round launch
- Update metadata at client context
Garbage Collection in COPS

• Metadata contains additional versions and dependencies
  • Reduce storage overhead

• Three kinds of GC for COPS & COPS-GT
  • Version GC
  • Dependency GC
  • Client Metadata GC
Garbage Collection in COPS

• Version GC (COPS-GT only)
  • Multiple versions of keys for COPS-GT
  • get_transaction algorithm limits the number of versions needed
  • After a new write, only keep the version around trans_time

• Dependency GC (COPS-GT only)
  • Dependencies for COPS-GT to get consistent view among keys
  • Dependencies can be removed once the version associated with old dependencies are no longer needed for correctness in get transaction

• Client Metadata GC (COPS + COPS-GT)
  • Client metadata (version, dependencies) stored in the client library
  • Client need to track dependencies only until they are guaranteed to be satisfied everywhere
  • Use flag called never-depend
Optimization in COPS – Nearest Dependencies

• Dependencies grow with client lifetime 😞

Transitively capture all ordering constraints
The nearest are few 😊

- Only check nearest when replicating
- COPS only tracks nearest
- COPS-GT tracks non-nearest for transactions
Fault Tolerance in COPS

• Assumption: Fail-stop & detectable

• Client failures
  • Simply stop issuing new requests & no recovery is necessary

• Key-Value node failures
  • Fault tolerant **linearizable** key-value store
    • FAWN-KV
  • **Chain replication** within cluster
    • Each data item is stored in a chain of R consecutive nodes along consistent hashing ring
    • `put_after` send to head & `get_by_version` send to tail

• Datacenter failures (partitions)
  • `put_after` in failed DC (may **permanently** loss data)
  • Replication queue in **active DC** will grow (need human work)
  • Dependency GC will **stop** (need wait or reconfiguration)
Outline

• Background & Motivation of COPS

• System Design of COPS

• Experiments & Evaluation

• Conclusion
Experimental Setup

• Hardware
  • 70 Linux servers
  • 2x6 core Intel Xeon X5650 CPUs, 48GB RAM, 3x1TB Hard Drives, and 2x1GigE network ports

• Software
  • FAWN-KV, Apache Thrift, Google Snappy

• Systems under comparison
  • LOG, COPS, COPS-GT

• Experiments
  • Microbenchmark
  • Dynamic workloads
  • Scalability

<table>
<thead>
<tr>
<th>System</th>
<th>Causal+</th>
<th>Scalable</th>
<th>Get Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOG</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>COPS</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>COPS-GT</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Microbenchmarks

• Simple setting: Two DCs, one server and one client per DC
• Saturate the system by put and get requests

<table>
<thead>
<tr>
<th>System</th>
<th>Operation</th>
<th>50%</th>
<th>99%</th>
<th>99.9%</th>
<th>Throughput (Kops/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thrift</td>
<td>ping</td>
<td>0.26</td>
<td>3.62</td>
<td>12.25</td>
<td>60</td>
</tr>
<tr>
<td>COPS</td>
<td>get_by_version</td>
<td>0.37</td>
<td>3.08</td>
<td>11.29</td>
<td>52</td>
</tr>
<tr>
<td>COPS-GT</td>
<td>get_by_version</td>
<td>0.38</td>
<td>3.14</td>
<td>9.52</td>
<td>52</td>
</tr>
<tr>
<td>COPS</td>
<td>put_after (1)</td>
<td>0.57</td>
<td>6.91</td>
<td>11.37</td>
<td>30</td>
</tr>
<tr>
<td>COPS-GT</td>
<td>put_after (1)</td>
<td>0.91</td>
<td>5.37</td>
<td>7.37</td>
<td>24</td>
</tr>
<tr>
<td>COPS-GT</td>
<td>put_after (130)</td>
<td>1.03</td>
<td>7.45</td>
<td>11.54</td>
<td>20</td>
</tr>
</tbody>
</table>

Latency for get_by_version is similar to ping using Thrift

Latency for put_after is slightly higher because puts are more expensive

Similar for throughput
Dynamic Workloads

- Dynamic workload with interacting clients accessing the COPS system
- Two DCs, S servers and S clients per DC
- The clients access storage servers in the local datacenter, which replicates put after operations to the remote datacenter
- Parameters

<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 1:4</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>

Keygroups are used to simulate the dependencies between operations
Dynamic Workloads (cont.)

- System throughput as a function of increasing delay between client operations

When the inter-operation delay is **low**, COPS **significantly** outperforms COPS-GT.

When the inter-operation delay is **high**, the maximum throughputs of COPS and COPS-GT **converge**.

As the inter-operation delay **increases**, the number of dependencies per operation **decreases**.

Garbage collection for dependencies!
Dynamic Workloads (cont.)

- System performance under varying put: get ratios and key-access distributions (variance)

Throughput increases for read-heavier workloads

COPS-GT becomes competitive with COPS for read-mostly workloads

The throughput of COPS-GT is affected by variance

Fewer dependencies translates to less metadata that needs to be propagated and thus higher throughput
Dynamic Workloads (cont.)

- Effect of keygroup size on the throughput of COPS and COPS-GT

COPS throughput decreases when number of keys in Keygroup increases

Decrease for COPS-GT: Inherit more dependencies in Get operation

Throughput of COPS is unaffected by different variances because Get operations in COPS never inherit extra dependencies

COPS-GT has an increased chance of inheriting dependencies as variance increases that decreased throughput.

As the size of values increases, the relative throughput of COPS-GT approaches that of COPS

Reason: As processing time per operation increases, the inter-operation delay increases, which in turn leads to fewer dependencies

Increase for COPS-GT: Garbage collection for dependencies
Scalability

- **LOG vs. COPS & COPS-GT**
  - **LOG**: 1 server/datacenter
  - **COPS & COPS-GT**: 1, 2, 4, 8, or 16 servers/datacenter

COPS and COPS-GT scale well in all scenarios!

Throughput of COPS and COPS-GT are very comparable!

COPS and COPS-GT scale out linearly!
Outline

• Background & Motivation of COPS

• System Design of COPS

• Experiments & Evaluation

• Conclusion
Conclusion

• Novel Properties
  • First ALPS and causal+ consistent system in COPS
  • Lock free, low latency get transactions in COPS-GT

• Novel techniques
  • Explicit dependency tracking and verification with decentralized replication
  • Optimizations to reduce metadata and checks

• COPS achieves high throughput and scales out
References


• [2] Lloyd W’s presentation video in SOSP 2011 for Don’t settle for eventual: scalable causal consistency for wide-area storage with COPS; Link: https://www.youtube.com/watch?v=jh9P1moDpAc

• [3] Lloyd W’s presentation slides in SOSP 2011 for Don’t settle for eventual: scalable causal consistency for wide-area storage with COPS; Link: https://pdfs.semanticscholar.org/81cb/3d879519b544b9250c020beb1ef14a61eebb.pdf