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

W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen

Presented by Zach Carey
Let's build a distributed data store!

Store items as key-value pairs

Desired operations:

Read a value based on key:    \(\text{val} = \text{get}(\text{key})\)
Write a value to a key:       \(\text{put}(\text{key}, \text{val})\)
Let's build a distributed data store!

Desired properties:

1. Availability
2. Network Partition Tolerance
3. Strong Consistency
A distributed data store **cannot** provide availability, network partition tolerance, **and** strong consistency.
Real Systems

Desired properties:

1. Availability
2. Network Partition Tolerance
3. Strong Consistency
Real Systems

Desired properties:

1. Availability
2. Network Partition Tolerance
3. No strong consistency...
"Eventual" Consistency

For a given key, replicas will *eventually* converge on the correct value.

Can we do better?
Consistency Options

- Linearizability
- Sequential
  - Per-Key Sequential
  - Eventual
- Causal
- FIFO

Strength
Consistency Options

- Linearizability
- Sequential
- Per-Key Sequential
- Causal
- Eventual
- FIFO

CAP Theorem...
Consistency Options

- Linearizability
- Sequential
- Per-Key Sequential
- Eventual
- Causal
- FIFO

Latency Requirements...
Consistency Options

- Linearizability
- Sequential
- Per-Key Sequential
- Eventual
- Causal
- FIFO

Want to do better...
Introduce Causal+

- Linearizability
- Sequential
- Causal+
- Per-Key Sequential
- Causal
- Eventual
- FIFO

Strength
Agenda

- Motivation
- Define Causal+
- COPS & COPS-GT
- Evaluation
- Conclusion + Discussion
Agenda

● Motivation

● **Define Causal+**

● COPS & COPS-GT

● Evaluation

● Conclusion + Discussion
Let $a \leadsto b$ denote that $b$ is potentially dependent on $a$. 
Rule 1: "Execution Thread"

If $a$ happens before $b$ on the same thread of execution, then $a \rightsquigarrow b$
Rule 2: "Gets From"

If $a$ is a put and $b$ is a get that returns the same value, then $a \sim b$
Rule 3: Transitivity

If $a \mapsto b$ and $b \mapsto c$, then $a \mapsto c$
Potential Causality $a \leadsto b$

1. **Execution Thread**: if $a$ happens before $b$ on the same thread of execution, then $a \leadsto b$

2. **Gets From**: if $a$ is a `put` and $b$ is a `get` that returns the same value, then $a \leadsto b$

3. **Transitivity**: if $a \leadsto b$ and $b \leadsto c$, then $a \leadsto c$
Causal Consistency

Values returned from `get` operations at a replica are consistent with the order defined by \( \rightsquigarrow \).
Problem: Conflicts

Two put operations to the same key that are not causally related

Client A

\[ x = 0 \]

Client B

\[ x = 0 \]
Problem: Conflicts

Two \texttt{put} operations to the same key that are not causally related

Client A \hspace{1cm} \texttt{put}(x,1) \hspace{1cm} x = 0

Client B \hspace{1cm} \texttt{put}(x,2) \hspace{1cm} x = 0
Problem: Conflicts

Two put operations to the same key that are not causally related

Client A

Client B
Problem: Conflicts

Two put operations to the same key that are not causally related

Client A

Client B

x = 1

x = 2

Replicas have diverged!
Causal+ Consistency

Values returned from `get` operations at a replica are consistent with the order defined by \( \sim \) with convergent conflict resolution.
COPS Overview

Client Library

Local Datacenter

Node

Other Datacenters
Client Library Interface

Read a value based on key: \[ \text{val} = \text{get}(\text{key}, \text{ctx}) \]

Write a value to a key: \[ \text{put}(\text{key}, \text{val}, \text{ctx}) \]

Create context: \[ \text{ctx} = \text{createContext}() \]

Delete context: \[ \text{bool} = \text{deleteContext}(\text{ctx}) \]
Client Library Storage

The client library will be storing <key, version> pairs.

On a get, retrieved <key, version> pair is added.

On a put, entries are cleared and replaced with this put.
Datacenter Interface

```
put_after(key, value, nearest, version)

<value, version> = get_by_version(key, version)
```
Conflict Detection

Invoke the "last-writer-wins" rule with the version number

- Use Lamport Timestamp

Diagram:
- Lamport Timestamp
  - Lamport Clock
  - Unique Node ID
Example put

1. Client calls `put(key, val)`
Example $\text{put}$

2. Client Library calculates nearest dependency

<table>
<thead>
<tr>
<th>key</th>
<th>version</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>1</td>
</tr>
<tr>
<td>z</td>
<td>2</td>
</tr>
</tbody>
</table>
Example **put**

3. Client Library sends **put_after** request

<table>
<thead>
<tr>
<th>key</th>
<th>version</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>1</td>
</tr>
<tr>
<td>z</td>
<td>2</td>
</tr>
</tbody>
</table>
Example put

4. Return new version

<table>
<thead>
<tr>
<th>key</th>
<th>version</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>1</td>
</tr>
<tr>
<td>z</td>
<td>2</td>
</tr>
</tbody>
</table>
Example **put**

5. Client Library updates metadata and returns

```
Client Library:
key: x
version: 3
```

SUCCESS
Example \texttt{put}

6. Local Datacenter forwards to others

\begin{center}
\begin{tikzpicture}[>=latex, auto]
    
    \node[draw, ellipse, fill=red!30] (node1) at (0,0) {Node};
    \node[draw, ellipse, fill=blue!30] (node2) at (0,-2) {Primary Node};
    \node[draw, ellipse, fill=gray!30] (node3) at (0,-4) {Node};
    \node[draw, ellipse, fill=red!30] (node4) at (4,0) {Node};
    \node[draw, ellipse, fill=blue!30] (node5) at (4,-2) {Primary Node};
    \node[draw, ellipse, fill=gray!30] (node6) at (4,-4) {Node};

    \path (node1) edge[<-] node {$\texttt{put\_after}(x, 4, \langle z, 2 \rangle, 3)$} (node5);

    \end{tikzpicture}
\end{center}
Example get

1. Client calls $\text{get}(\text{key})$
Example \textit{get}

2. Client Library sends \textit{get\_by\_version} request

\begin{table}[h]
\begin{tabular}{|c|c|}
\hline
key & version \\
\hline
x & 3 \\
\hline
\end{tabular}
\end{table}
Example \textit{get}

3. \textbf{Returns} \textit{<value, version>}

<table>
<thead>
<tr>
<th>key</th>
<th>version</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>3</td>
</tr>
</tbody>
</table>
4. Client Library updates metadata and returns

<table>
<thead>
<tr>
<th>key</th>
<th>version</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>4</td>
</tr>
</tbody>
</table>
We can write values with put.

We can retrieve values with get.

These operations respect causal+ consistency.

But there is still in issue...
Consistent Dependent \texttt{get} Requests

Let $x$ and $y$ be dependent keys

Client A

\texttt{put}(x,0) \rightarrow \texttt{put}(y,0) \rightarrow \texttt{put}(x,1) \rightarrow \texttt{put}(y,1)

Client B

$0=\texttt{get}(x) \rightarrow 1=\texttt{get}(y)$

These values are inconsistent with each other
Client Library Interface

Read a value based on key: \( \text{val} = \text{get}(\text{key}, \text{ctx}) \)

Write a value to a key: \( \text{put}(\text{key}, \text{val}, \text{ctx}) \)

Create context: \( \text{ctx} = \text{createContext}() \)

Delete context: \( \text{bool} = \text{deleteContext}(\text{ctx}) \)

Get collection of keys: \( \text{<values>} = \text{get\_trans}(\text{<keys>}, \text{ctx}) \)
COPS-GT Client Library Changes

The client library will be storing \(<key, \text{version}, \text{dep}>\) tuples.

On a **get**, retrieved \(<key, \text{version}, \text{dep}>\) tuple is added.

On a **put**, that key's deps are set to all other keys in that context.
Datacenter Interface Changes

```
put_after(key, value, [deps], nearest, version)

<value, version, deps> = get_by_version(key, version)
```
Example $\text{put} \ (\text{COPS-GT})$

1. Client calls $\text{put}(\text{key}, \text{val})$
Example $\text{put } (\text{COPS-GT})$

2. Client Library calculates all dependencies
Example \texttt{put} (COPS-GT)

3. Client Library sends \texttt{put\_after} request

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
key & version & deps \\
\hline
y & 1 & \ \\
\hline
z & 2 & \textless y,1\textgreater  \\
\hline
\end{tabular}
\end{center}
Example \texttt{put (COPS-GT)}

4. Return new version

\begin{tabular}{|c|c|c|}
  \hline
  key & version & deps \\
  \hline
  y   & 1       & \textless y,1\textgreater \\
  z   & 2       & \textless y,1\textgreater \\
  \hline
\end{tabular}
Example \texttt{put} (COPS-GT)

5. Client Library updates metadata and returns

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
key & version & deps \\
\hline
y & 1 & \\
\hline
z & 2 & \textless{}y,1\textgreater{} \\
\hline
x & 3 & \textless{}y,1\textgreater{}, \textless{}z,2\textgreater{} \\
\hline
\end{tabular}
\end{table}
Example $\text{put (COPS-GT)}$

6. Local Datacenter forwards to others

Local Datacenter

Node

Primary Node

Node

$\text{put\_after(x, 4, deps, 3)}$

Primary Node

Node

Node
Example `get (COPS-GT)`

1. Client calls `get (key)`
Example \texttt{get (COPS-GT)}

2. Client Library sends \texttt{get\_by\_version} request

<table>
<thead>
<tr>
<th>key</th>
<th>version</th>
<th>deps</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>2</td>
<td>\langle y, 1 \rangle</td>
</tr>
<tr>
<td>x</td>
<td>3</td>
<td>\langle y, 1 \rangle, \langle z, 2 \rangle</td>
</tr>
</tbody>
</table>
Example `get (COPS-GT)`

3. Returns `<value, version, deps>`

```
key | version | deps
--- | ------- | ----
y  | 1       |     
z  | 2       | <y,1>
x  | 3       | <y,1> <z,2>
```

Local Datacenter
- Node
- Primary Node
Example `get (COPS-GT)`

4. Client Library updates metadata and returns

<table>
<thead>
<tr>
<th>key</th>
<th>version</th>
<th>deps</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>4</td>
<td><code>&lt;z, 2&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;x, 3&gt;</code></td>
</tr>
<tr>
<td>z</td>
<td>2</td>
<td><code>&lt;y, 1&gt;</code></td>
</tr>
<tr>
<td>x</td>
<td>3</td>
<td><code>&lt;y, 1&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;z, 2&gt;</code></td>
</tr>
</tbody>
</table>
COPS-GT Get Transaction: Two Rounds

Round 1:

- **Issue a get_by_version for each key concurrently**
- **Check dependencies. Satisfied if:**
  - Dependency was not in the request
  - OR Key was requested, and its version is ≥ dependency

Example:

<table>
<thead>
<tr>
<th>Value Requested</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Dependencies</td>
<td>&lt;Z, 5&gt;</td>
<td>&lt;X, 4&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;Y, 1&gt;</td>
<td></td>
</tr>
</tbody>
</table>
COPS-GT Get Transaction: Two Rounds

Round 2

- For each inconsistent key, call `get_by_version` again

In this example, request version 4 of X

<table>
<thead>
<tr>
<th>Value Requested</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Dependencies</td>
<td>&lt;Z, 5&gt;</td>
<td>&lt;X, 4&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;Y, 1&gt;</td>
<td></td>
</tr>
</tbody>
</table>
Let's revisit this example

Let $x$ and $y$ be dependent keys

Client A
- put($x$, 0) → put($y$, 0) → put($x$, 1) → put($y$, 1)

Client B
- 0 = get($x$)
- 1 = get($y$)

These values are inconsistent with each other
Let's revisit this example

Let $x$ and $y$ be dependent keys

Client A  $\xrightarrow{\text{put}(x, 0)}$  Client B  $\xrightarrow{\text{put}(y, 0)}$  $\xrightarrow{\text{put}(x, 1)}$  $\xrightarrow{\text{get\_trans}(x, y)}$  $\xrightarrow{\text{put}(y, 1)}$
Round 1:

**Call** `get_by_version` **for X and Y**

<table>
<thead>
<tr>
<th>Value Requested</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Dependencies</td>
<td></td>
<td>&lt;X, 3&gt;</td>
</tr>
</tbody>
</table>
Round 2:

Call `get_by_version(x, 3)` to satisfy Y's dependencies

Problem Solved!
Garbage Collection

Define a timeout $T$ for $\text{get\_trans}$

1. Key Versions: Clean up after $T$ seconds

2. Dependencies: Clean up $T$ seconds after all data centers commit value

3. Client Metadata: Clean up when Datacenter communicates:
   a. A "never-depend" flag
   b. Global Checkpoint Time
Fault Tolerance

1. Client Failures: Do nothing

2. Node Failures: Chain Replication

3. Datacenter Failures:
   a. put_after operations lost / delayed
   b. Garbage Collection: Fix Partition or System reconfiguration
Conflict Detection

Invoke the "last-writer-wins" rule with the version number

- Use Lamport Timestamp

Diagram:
- Lamport Timestamp
  - Lamport Clock
  - Unique Node ID
Evaluation: Scalability

Throughput vs Number Servers per Datacenter

Doubles as servers double

Throughput (Kops/s)

Number of Servers

COPS

COPS-GT

LOG
### Evaluation: Latency

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

Throughput (Kops/s)

Put: Get Ratio

Key Variance
Conclusion

- Distributed data stores should strive for higher consistency than the eventual consistency model
- COPS & COPS-GT are scalable implementations of causal+ consistency
Thank you!
Discussion