Don't Settle for Eventual: Scalable Causal Consistency with COPS

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen

Presented by Ben Reeves
Large-scale systems need geo-replicated storage

Image source: https://queue.acm.org/detail.cfm?id=2610533
Distributed storage: desirable properties

Strong Consistency: *I see what you see.*

Availability: *I can see it.*

Partition Tolerance: *We can see it, even if we can't speak.*
You-pick-two: Bayou

Strong Consistency:  I see what you see.

Availability:  I can see it.

Partition Tolerance:  We can see it, even if we can't speak.
You-pick-two: Bayou

Strong Consistency: I see what you see.

Availability: I can see it.

Partition Tolerance: We can see it, even if we can't speak.

“We believe that applications must be aware that they may read weakly consistent data.”
You-pick-two: Dynamo

Strong Consistency: I see what you see.

Availability: I can see it.

Partition Tolerance: We can see it, even if we can't speak.
You-pick-two: Dynamo

**Strong Consistency:** I see what you see.

**Availability:** I can see it.

**Low Latency:** I can see it quickly.

**Partition Tolerance:** We can see it, even if we can't speak.

**Scalability:** There's lots of us and lots of it.
You-pick-two: Google’s Spanner

Strong Consistency:  I see what you see.

Availability:  I can see it.

Low Latency:  I can see it quickly.

Partition Tolerance:  We can see it, even if we can't speak.

Scalability:  There’s lots of us and lots of it.
You-pick-two: Google's Spanner

**Strong Consistency:** *I see what you see.*

**Availability:** *I can see it.*

**Low Latency:** *I can see it quickly.*

**Partition Tolerance:** *We can see it, even if we can't speak.*

**Scalability:** *There's lots of us and lots of it.*
Distributed storage: the status quo

**Availability:**  
*I can see it.*

**Low Latency:**  
*I can see it quickly.*

**Partition Tolerance:**  
*We can see it, even if we can't speak.*

**Scalability:**  
*There’s lots of us and lots of it.*

Weak Consistency / Eventual Consistency
The importance of stronger consistency: exhibit A

Alice: “I lost my dog!”

Alice: “He came back!”

Bob: “That’s great!”

US-East

US-West
The importance of stronger consistency: exhibit A

Alice: “I lost my dog!”
Alice: “He came back!”
Bob: “That’s great!”

US-East

US-West

Alice: “I lost my dog!”
Bob: “That’s great!”
Alice: “He came back!”
Bob: “That’s great!”
Student **deletes** incriminating status.

Student **accepts** advisor’s friend request.

Advisor can **see** incriminating status.

Advisor can only see innocent photos.
The importance of stronger consistency: exhibit C

Alice *uploads* a photo.

Alice *creates* a new album.

Alice *adds* photo to album.

US-East

US-West

No photo or album exists.

Album exists, is empty.

Album remains empty.
Causal consistency to the rescue

Causal consistency:

Each **write** operation must appear as though it occurs **after** all operations that **causally precede** it.
1. **Execution thread**: $a \rightarrow b$ within a single thread.

Client 1 \hspace{1cm} \text{put}(x, 2) \rightarrow \text{put}(y, 2)

Client 2

Time
2. Reads from: 

\[ \text{put}(x, 2) \rightarrow \text{put}(y, 2) \]

\[ \text{get}(y) = 2 \]

\[ b \text{ reads what } a \text{ writes.} \]
3. **Transitivity:** \( a \rightarrow b \) and \( b \rightarrow c \), so \( a \rightarrow c \)

Client 1: \( \text{put}(x, 2) \rightarrow \text{put}(y, 2) \)

Client 2: \( \text{get}(y) = 2 \)
Causal consistency in practice

Some previous systems provide causal consistency:

- PRACTI
- Bayou’s anti-entropy protocol
- others...

None provide causal consistency scalably.
Causal consistency in COPS
COPS: networking model

US-East

EU (Paris)

Asia Pacific
COPS: networking model

Data Center (Replica)

Frontend COPS clients

Linearizeable KV store
COPS: the general idea

All reads and writes executed at **local replica first**.

All writes **replicated** to other replicas **asynchronously**.

At remote replicas:

**Wait** until dependencies satisfied before replicating writes.
Representing causal dependencies in COPS

\[ \text{put}(x, 2) \rightarrow \text{put}(y, 2) \rightarrow \text{put}(x, 4) \]
COPS storage api

<val, version> ← get(key)

<bool, version> ← put_after(key, val, deps)
COPS storage api

\[
\begin{align*}
\text{<val, version>} & \leftarrow \text{get(key)} \\
\text{<bool, version>} & \leftarrow \text{put\_after(key, val, deps)} \\
\text{bool} & \leftarrow \text{dep\_check(key, version)}
\end{align*}
\]
Ensuring causal consistency in COPS

put(status, "sleep")

put(friends, "lorenzo")

status = "illegal things"

Advisor can see bad status.
Ensuring causal consistency in COPS

\[
\text{put\_after}(S_2, \text{“sleep”}, [])
\]

\[
\text{put\_after}(F_1, \text{“lorenzo”}, [S_2])
\]

\[S_1 = \text{“illegal”}\]
Ensuring causal consistency in COPS

\[
\text{put_after}(S_2, \text{“sleep”}, [])
\]

\[
\text{put_after}(F_1, \text{“lorenzo”}, [S_2])
\]

\[
S_1 = \text{“illegal”}
\]

\[
\text{false} \leftarrow \text{dep\_check}(S_2)
\]
Ensuring causal consistency in COPS

\[
\text{put\_after}(S_2, \text{"sleep"}, [])
\]

\[
\text{put\_after}(F_1, \text{"lorenzo"}, [S_2])
\]

\[
S_1 = \text{"illegal"}
\]

false ← dep\_check(S_2)

true ← dep\_check(S_2)
Ensuring causal consistency in COPS

```
put_after(S2, "sleep", [])
put_after(F1, "lorenzo", [S2])
```

\[ S_1 = "illegal" \]

\[ \text{false} \leftarrow \text{dep\_check}(S_2) \]

\[ \text{true} \leftarrow \text{dep\_check}(S_2) \]
Tracking causal dependencies in COPS

**Client context** represents one execution thread.

Context stores current set of dependencies.

Each get/put adds a dependency to the set.
Tracking causal dependencies in COPS

**Client context** represents one execution thread.

```
ctx = createContext();
“foo” ← get(x, ctx)  --->  “foo”, x₁ ← get(x)
put(z, “foo”, ctx);  --->  true, z₁ ← put_after(z, “foo”, [x₁])
put(x, “bar”, ctx);  --->  true, x₂ ← put_after(x, “bar”, [x₁, z₁])
deleteContext(ctx);
```
Tracking causal dependencies in COPS

Only need to provide the nearest dependencies.

```javascript
ctx = createContext();
“foo” ← get(x, ctx)  --->  “foo”, x₁ ← get(x)
put(z, “foo”, ctx);  --->  true, z₁ ← put_after(z, “foo”, [x₁])
put(x, “bar”, ctx);  --->  true, x₂ ← put_after(x, “bar”, [x₁, z₁])
deleteContext(ctx);
```
Tracking causal dependencies in COPS

Only need to provide the nearest dependencies.

\[ t_2 \leftarrow \text{get}(t, \text{ctx}) \]
\[ u_1 \leftarrow \text{get}(u, \text{ctx}) \]
\[ v_6 \leftarrow \text{put}(v, \text{"foo"}) \]
Tracking causal dependencies in COPS

Only need to provide the **nearest dependencies**.

\[
\begin{align*}
t_2 &\leftarrow \text{get}(t, \text{ctx}) \\
u_1 &\leftarrow \text{get}(u, \text{ctx}) \\
v_6 &\leftarrow \text{put}(v, \text{“foo”}) \\
z_4 &\leftarrow \text{put}(z, \text{“bar”})
\end{align*}
\]
Tracking causal dependencies in COPS

Only need to provide the nearest dependencies.

t_2 \leftarrow \text{get}(t, \text{ctx})
u_1 \leftarrow \text{get}(u, \text{ctx})
v_6 \leftarrow \text{put}(v, \text{"foo"})
z_4 \leftarrow \text{put}(z, \text{"bar"})
Causal consistency limitation: conflicts

Provides no guarantees for **concurrent** writes.

Allows replicas to diverge indefinitely.

Client 1  \[\text{put}(x, 2) \rightarrow \text{put}(y, 2) \rightarrow \text{put}(x, 4)\]

Client 2  \[\text{get}(y) = 2 \rightarrow \text{put}(x, 3)\]
Causal+ consistency:

Causal consistency with convergent conflict handling.
Detecting conflicts in COPS-CD

Client 1  \text{put}(x, 3) \rightarrow \text{put}(x, 4) \quad \text{No conflict.}

Client 1  \text{put}(x, 4)

Client 2  \text{put}(x, 3) \quad \text{Conflict.}
Detecting conflicts in COPS-CD

Specify the **previous version** in all writes:

\[
\langle \text{bool}, \text{version} \rangle \leftarrow \text{put\_after}(\text{key}, \text{val}, \text{deps}, prev)
\]

Iff \(prev \neq curr\), then the write is conflicting.
Detecting conflicts in COPS-CD

Iff $prev \neq curr$, then the write is conflicting.

Client 1

- $\text{put\_after}(x_1, 3, \text{prev}=0)$
- $\text{put\_after}(x_2, 4, \text{prev}=1)$

Client 1

$\text{put\_after}(x_1, 3, \text{prev}=0)$

Client 2

$\text{put\_after}(x_1, 4, \text{prev}=0)$
COPS-CD is causal+ consistent and fast!
...but that’s not quite enough
The need for get-transactions

Alice sets album to **private**.

Alice **uploads** a photo.

```
put_after(acl_2, "private", [])
put_after(album_2, "photo", [acl_2])
```
The need for get-transactions

Alice sets album to **private**.

Alice **uploads** a photo.

Bob wants to view Alice’s album:

1. `public ← get(albumACL)`

2. `photo ← get(album)`

3. serve photo to Bob
The need for get-transactions

Alice sets album to \textbf{private}.

Alice \textbf{uploads} a photo.

Bob wants to view Alice’s album:

\begin{itemize}
  \item \textbf{public}, \textbf{empty} $\leftarrow$ \texttt{get\_trans(acl, album)}
  \item don’t serve Bob the photo
  \item \textbf{private}, \textbf{photo} $\leftarrow$ \texttt{get\_trans(acl, album)}
  \item don’t serve Bob the photo
\end{itemize}
How to achieve a get-transaction?

Alice sets album to **private**.

Alice **uploads** a photo.

Bob wants to view Alice’s album:

1. \[ \text{public} \leftarrow \text{get(acl}_1\right) \]

   \[ \text{put}_\text{after(acl}_2, \text{"private"}, []) \]

   \[ \text{put}_\text{after(album}_2, \text{"photo"}, [\text{acl}_2]) \]

2. \[ \text{photo} \leftarrow \text{get(album}_2\right) \]
How to achieve a get-transaction?

Alice sets album to **private**.

Alice **uploads** a photo.

Bob wants to view Alice’s album:

1. **public** ← get(acl₁)
   
   put_after(acl₂, "private", [])
   
   put_after(album₂, "photo", [acl₂])

2. **photo** ← get(album₂)

Bob needs to retrieve \([acl₂]\).
COPS-GT: supporting get transactions

Store dependencies for each <key, version>.

<table>
<thead>
<tr>
<th>Key</th>
<th>Vrs</th>
<th>Deps</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>acl</td>
<td>2</td>
<td>[]</td>
<td>private</td>
</tr>
<tr>
<td>album</td>
<td>2</td>
<td>[acl₂]</td>
<td>photo</td>
</tr>
</tbody>
</table>
COPS-GT: supporting get transactions

<val, vers> ← get(key)
<val, vers, deps> ← get(key)
COPS-GT: supporting get transactions

\[ \langle \text{val, vers} \rangle \leftarrow \text{get}(\text{key}) \]

\[ \langle \text{val, vers, deps} \rangle \leftarrow \text{get}_\text{by_version}(\text{key}, \text{vers}=\text{latest}) \]
COPS-GT: supporting get transactions

\[
<\text{bool}, \text{vers}> \leftarrow \text{put}\_\text{after}(k, v, \text{deps}, \text{prev})
\]

\[
<\text{bool}, \text{vers}> \leftarrow \text{put}\_\text{after}(k, v, \text{deps}, \text{nearest}, \text{prev})
\]

<table>
<thead>
<tr>
<th>Key</th>
<th>Vrs</th>
<th>Deps</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>acl</td>
<td>1</td>
<td>[]</td>
<td>public</td>
</tr>
<tr>
<td>acl</td>
<td>2</td>
<td>[]</td>
<td>private</td>
</tr>
<tr>
<td>album</td>
<td>1</td>
<td>[acl]</td>
<td>empty</td>
</tr>
<tr>
<td>album</td>
<td>2</td>
<td>[acl]</td>
<td>photo</td>
</tr>
</tbody>
</table>

The stored entries are:

- acl
- album

Put after these are satisfied.
COPS-GT: get transaction algorithm

<val, version> ← get_trans([keys...]):

**Round 1**

1. send out `get_by_version(k)` for each k in keys
2. collect all `<val_k, version_k, deps_k>`
3. combine all `deps_k` into D (take highest version of each)
4. for each `dep` in D
   a. if `key(dep)` in keys && `vers(dep) > version_{key(dep)}`
      i. `not_satisfied += dep`
COPS-GT: get transaction algorithm

\(<val, \text{version}> \leftarrow \text{get\_trans}([\text{keys}\ldots]):\)

Round 2 (if necessary)
1. for each \text{dep} in not\_satisfied:
    a. send \text{get\_by\_version}(\text{key}(\text{dep}), \text{vers}(\text{dep}))
COPS-GT: get transaction algorithm

ctxA = createContext()
put(acl, “private”, ctxA)
put(album, “photo”, ctxA)

ctxB = createContext()
get_trans([acl, album], ctxB)
COPS-GT: get transaction algorithm

cxA = createContext()
put(acl, "private", ctxA)
put(album, "photo", ctxA)

cxB = createContext()
get_trans([acl, album], ctxB)

get_by_version(acl)
get_by_version(album)

<table>
<thead>
<tr>
<th>Key</th>
<th>Vrs</th>
<th>Deps</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>acl</td>
<td>1</td>
<td>[]</td>
<td>public</td>
</tr>
<tr>
<td>album</td>
<td>1</td>
<td>[acl]</td>
<td>empty</td>
</tr>
</tbody>
</table>