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: `val = get(key)`

Write a value to a key: `put(key,val)`

Let's build a distributed data store!

Desired properties:

1. Availability

2. Network Partition Tolerance

3. Strong Consistency

CAP Theorem

*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...

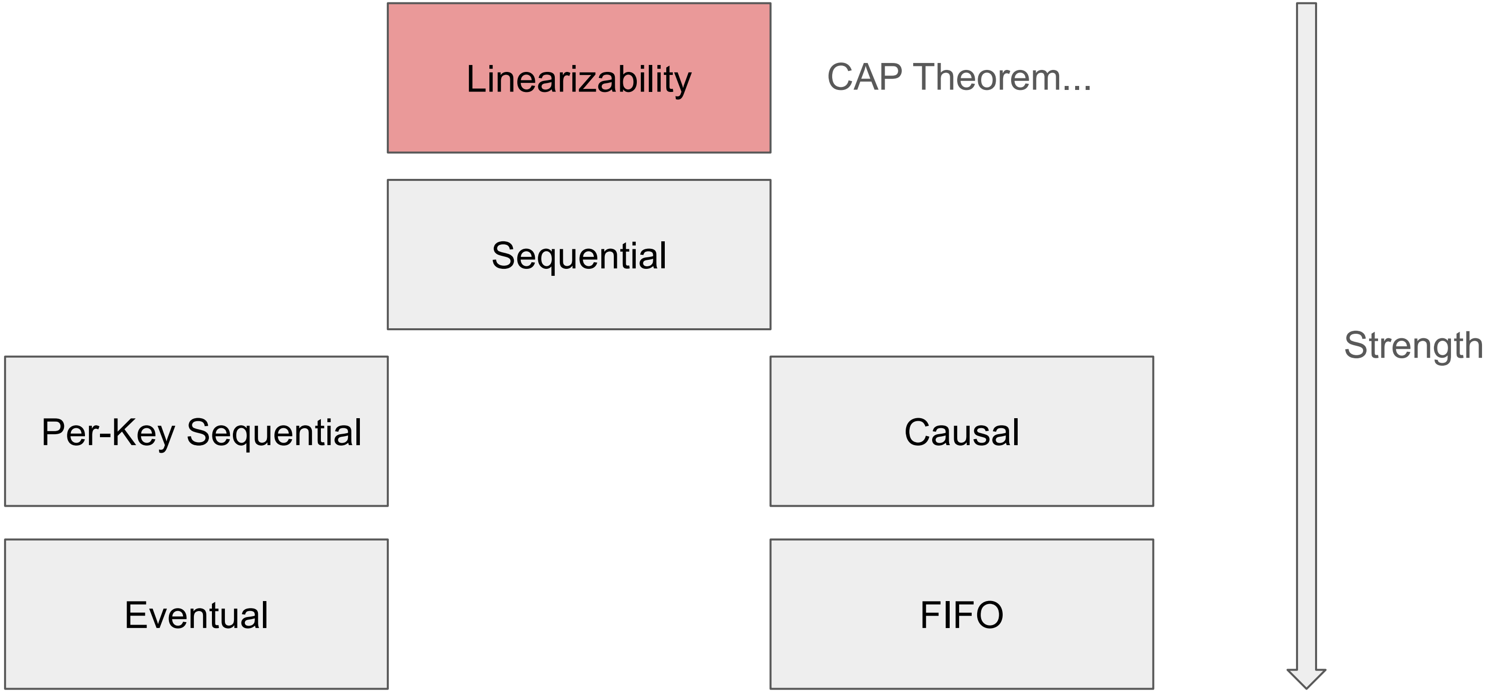
"Eventual" Consistency

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

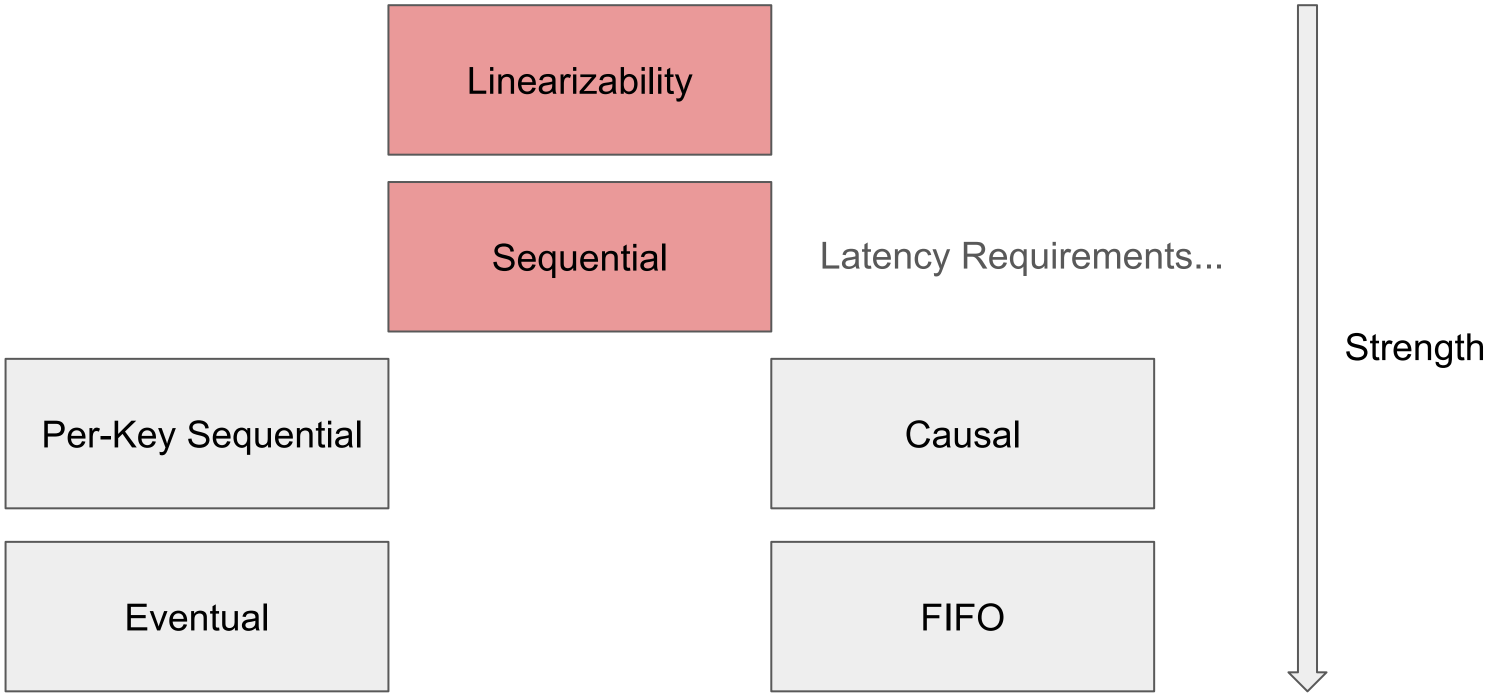
Can we do better?

Consistency Options

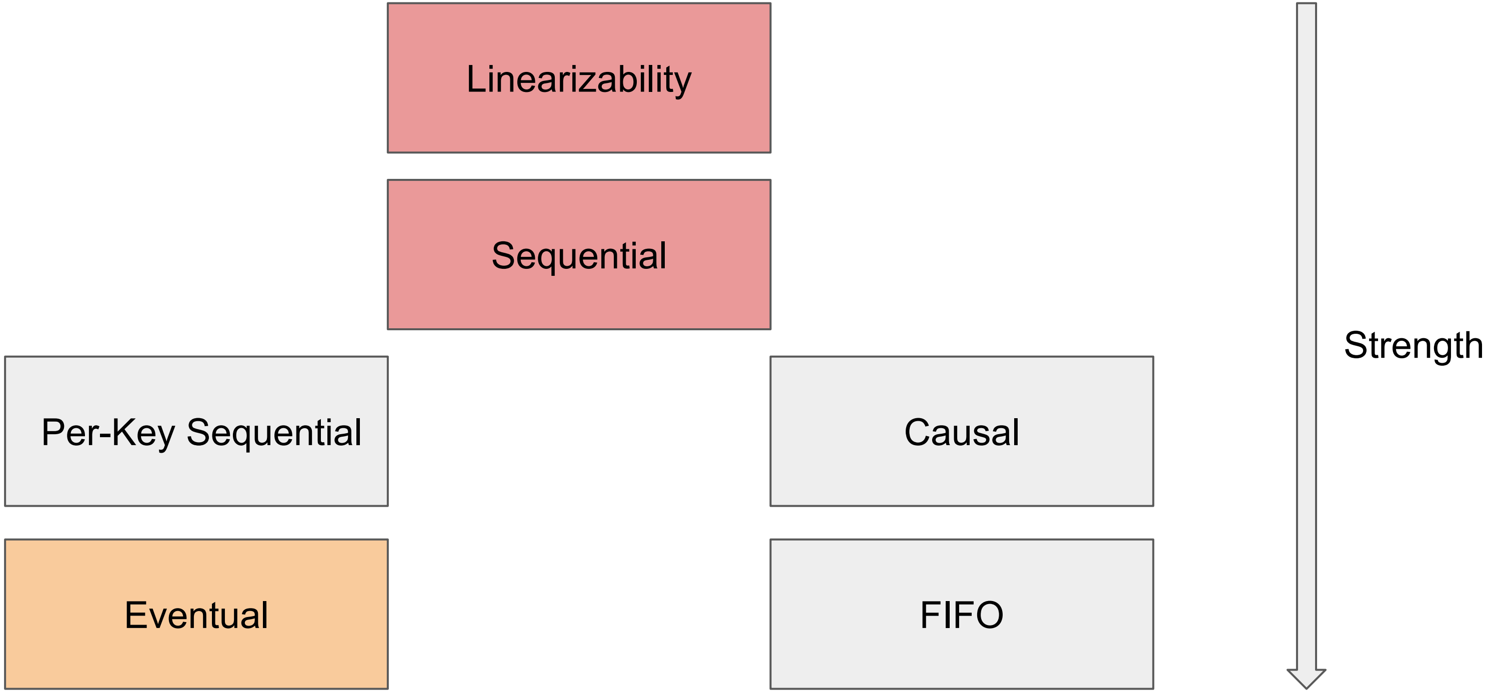
Consistency Options



Consistency Options

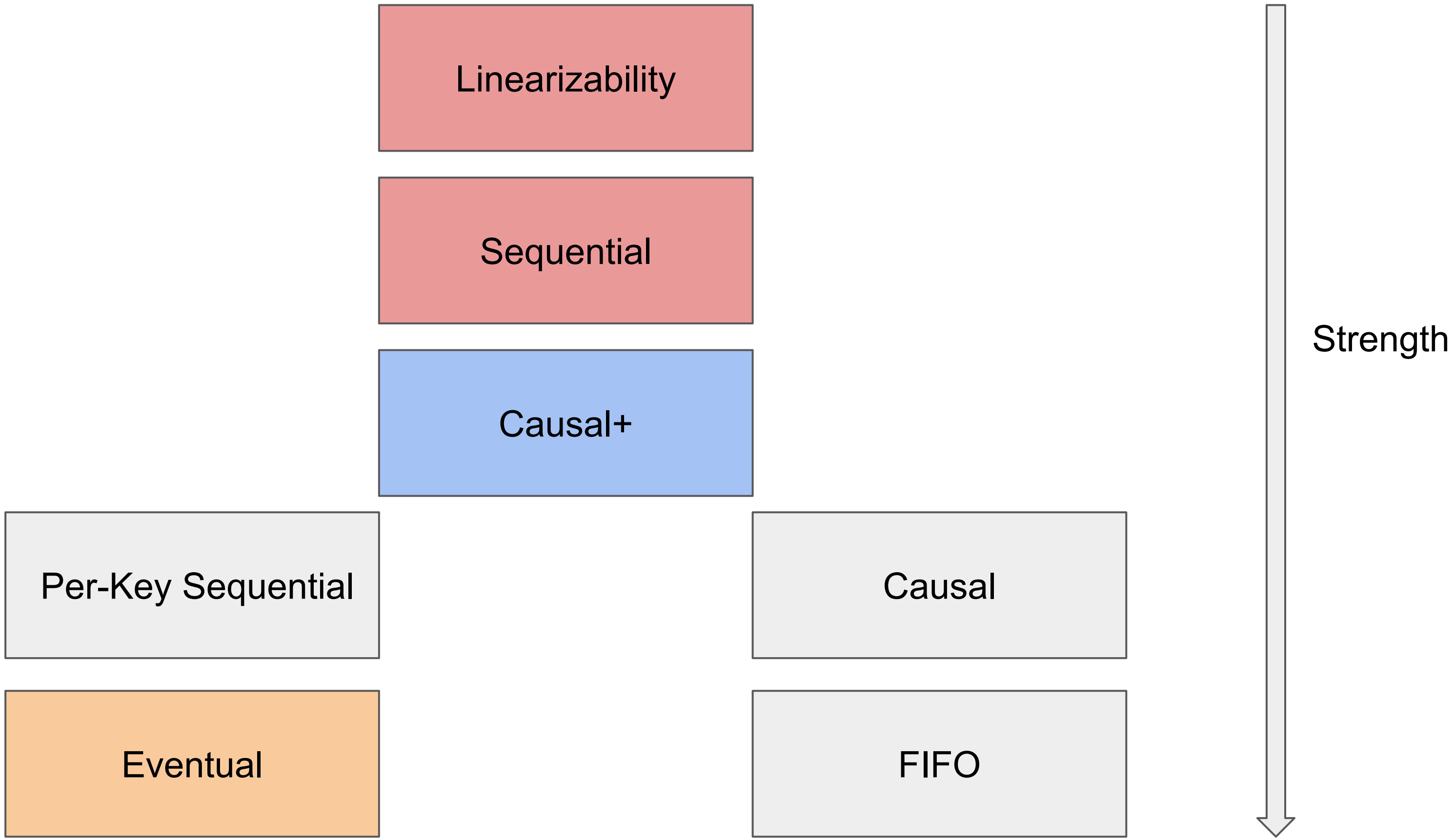


Consistency Options



Want to do better...

Introduce Causal+



Agenda

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

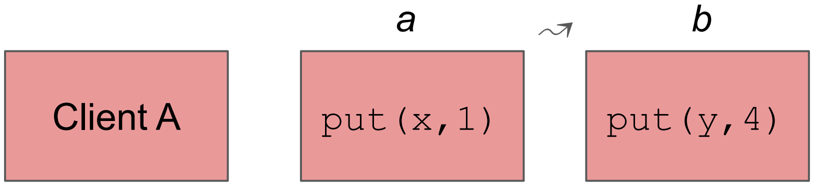
Agenda

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

Let $a \rightsquigarrow b$ denote that b is potentially dependent on a.

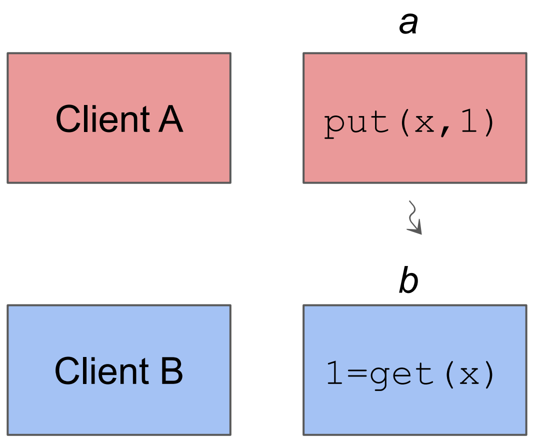
Rule 1: "Execution Thread"

If *a* happens before *b* on the same thread of execution, then $a \rightsquigarrow b$



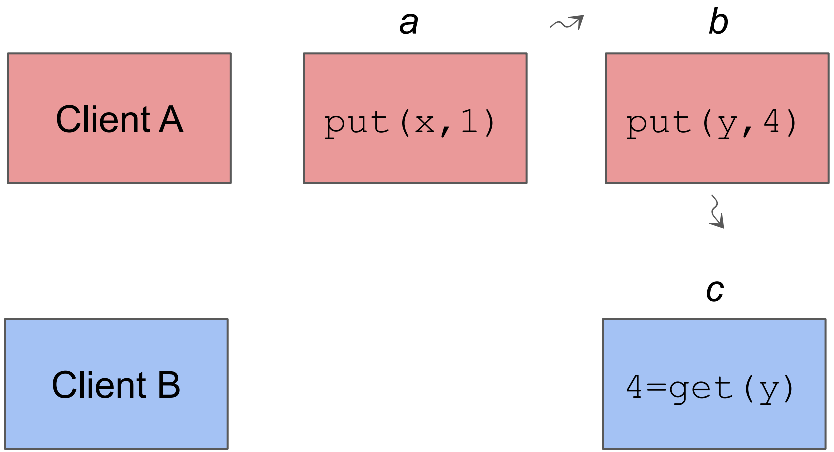
Rule 2: "Gets From"

If *a* is a `put` and *b* is a `get` that returns the same value, then $a \rightsquigarrow b$



Rule 3: Transitivity

If $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$



Potential Causality $a \rightsquigarrow b$

1. **Execution Thread**: if *a* happens before *b* on the same thread of execution, then $a \rightsquigarrow b$
2. **Gets From**: if *a* is a `put` and *b* is a `get` that returns the same value, then $a \rightsquigarrow b$
3. **Transitivity**: if $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$

Causal Consistency

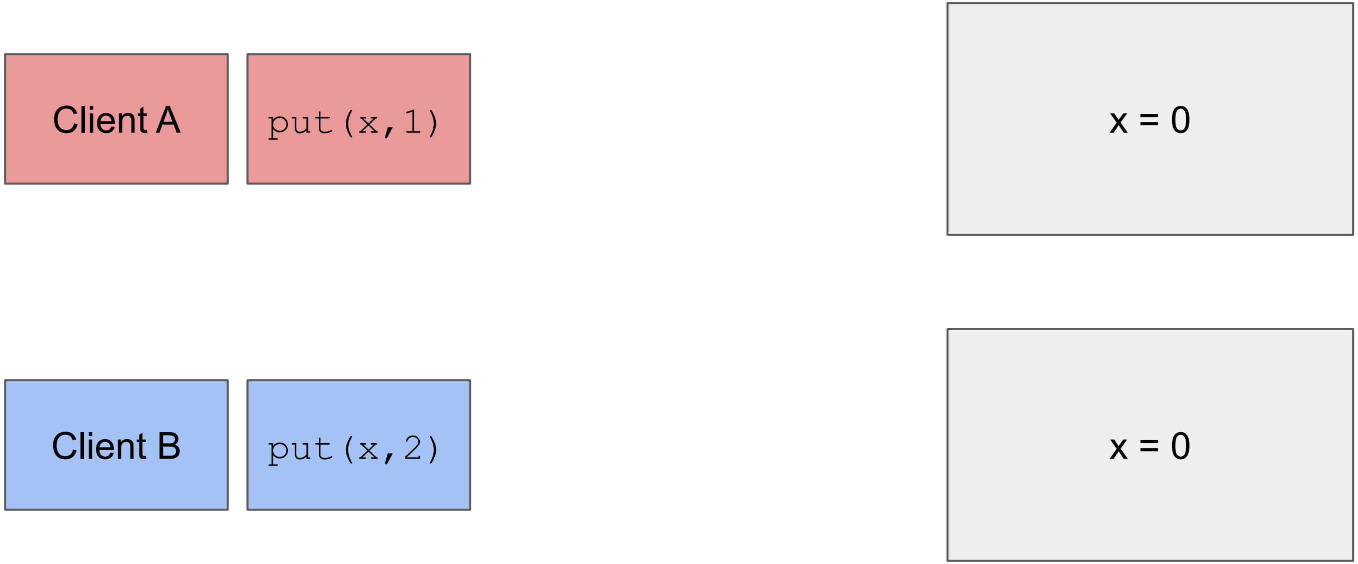
Values returned from `get` *operations at a replica are consistent with the order defined by* $\leadsto$

Problem: Conflicts

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

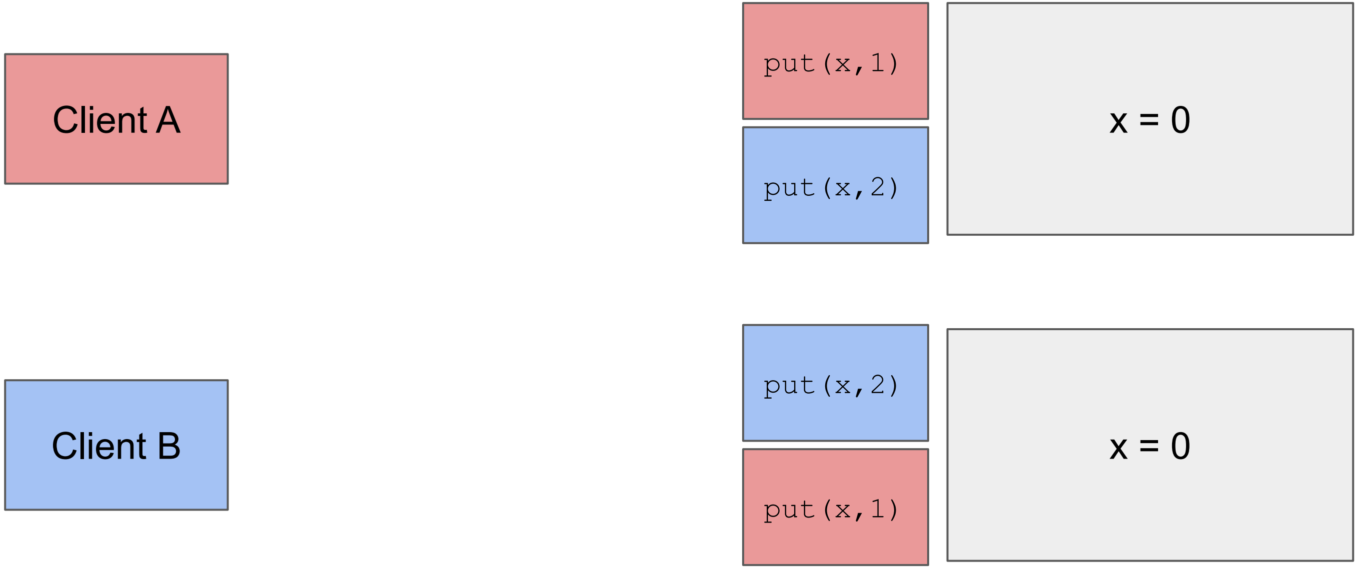
Problem: Conflicts

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



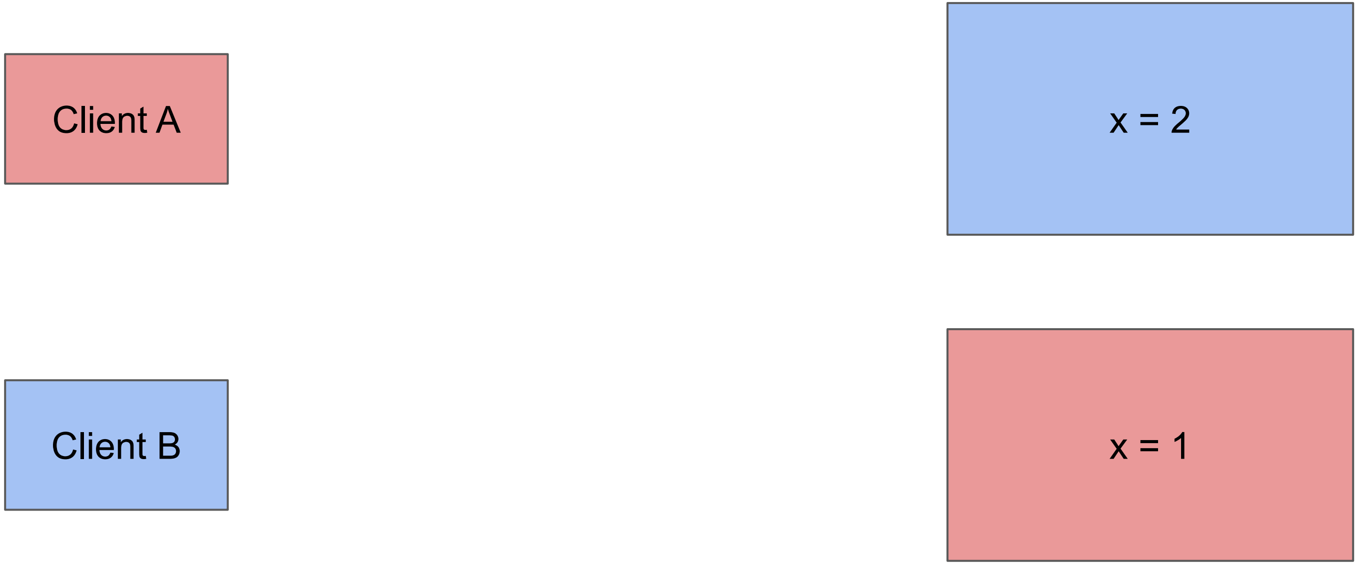
Problem: Conflicts

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



Problem: Conflicts

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

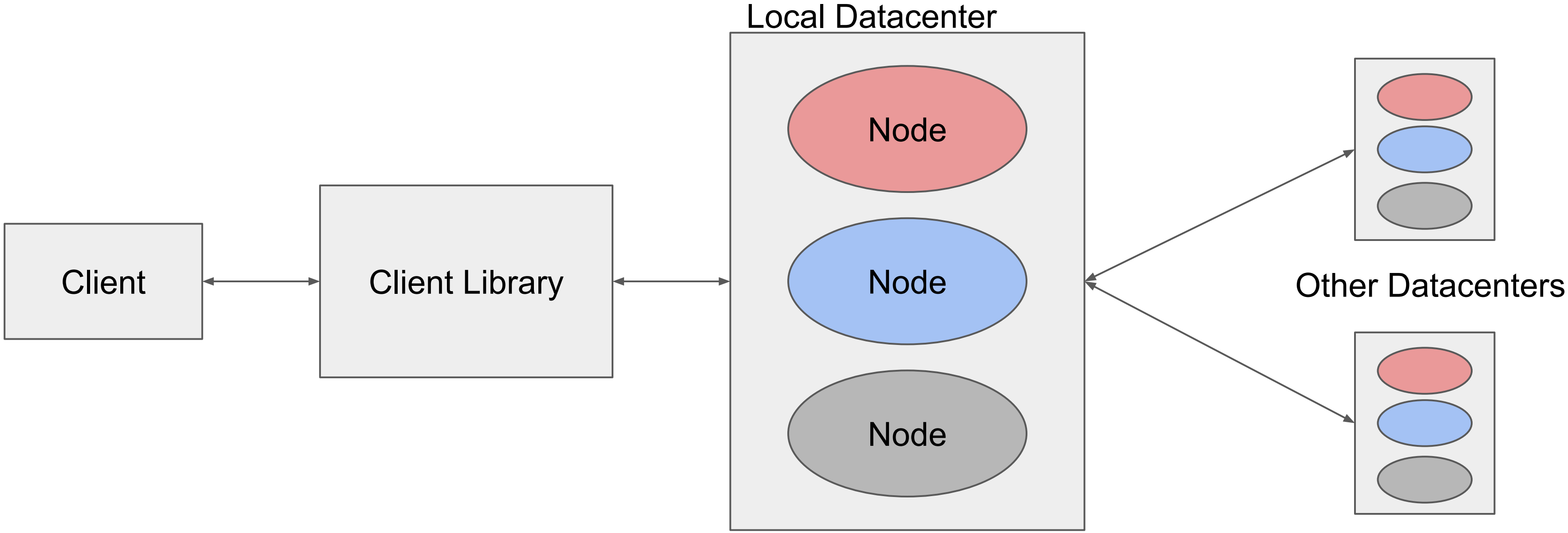


Replicas have diverged!

Causal+ Consistency

Values returned from `get` *operations at a replica are consistent with the order defined by* $\rightsquigarrow$ ***with convergent conflict resolution.***

COPS Overview



Client Library Interface

Read a value based on key: `val = get(key, ctx)`

Write a value to a key: `put(key, val, ctx)`

Create context: `ctx = createContext()`

Delete context: `bool = deleteContext(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

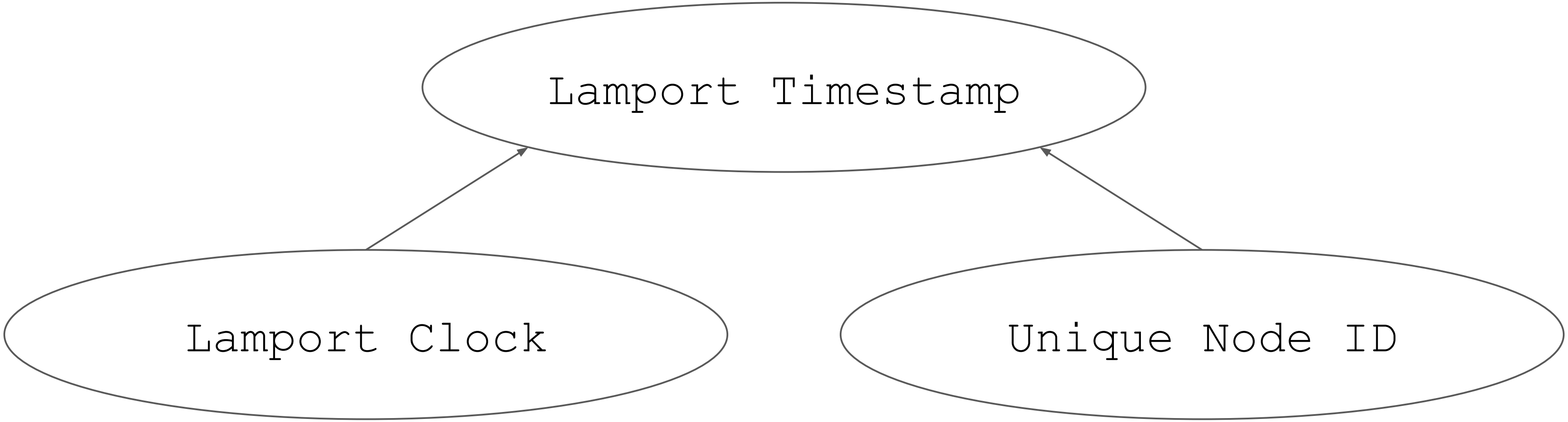
```
put_after(key, value, nearest, version)

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

Conflict Detection

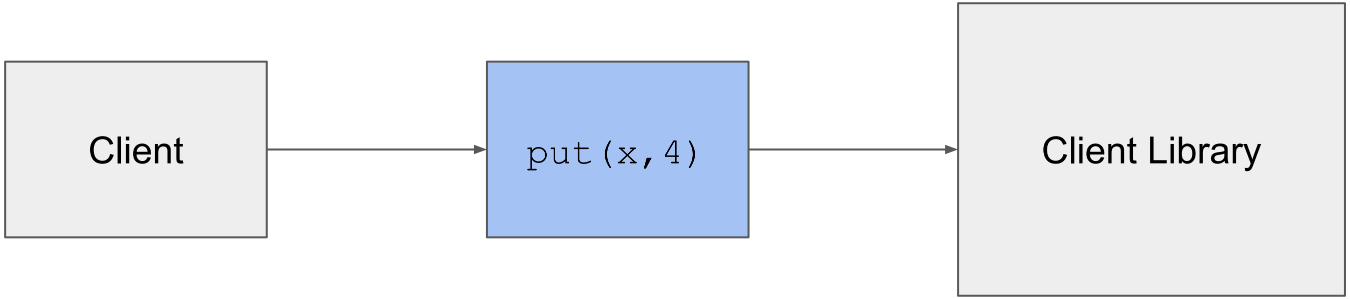
Invoke the "last-writer-wins" rule with the version number

- Use Lamport Timestamp



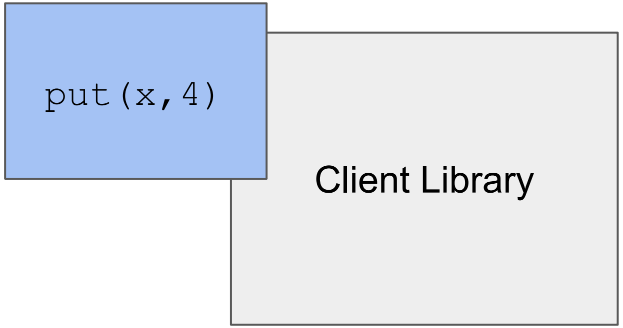
Example `put`

1. Client calls `put(key, val)`



Example `put`

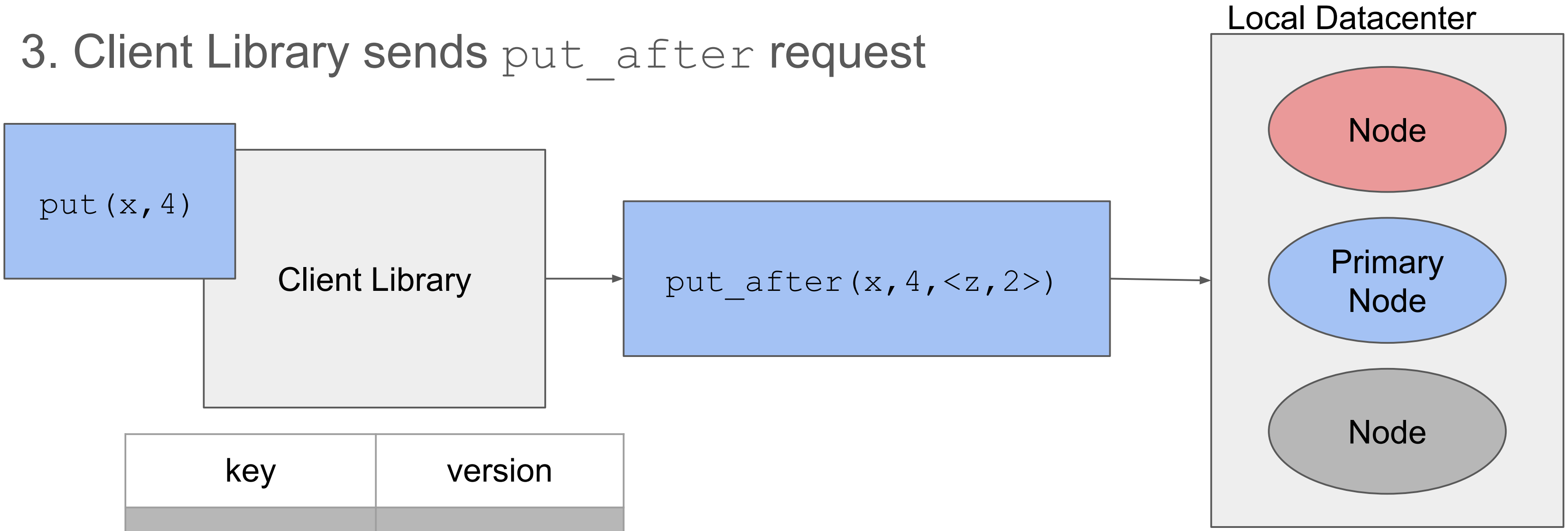
2. Client Library calculates nearest dependency



key	version
y	1
z	2

Example `put`

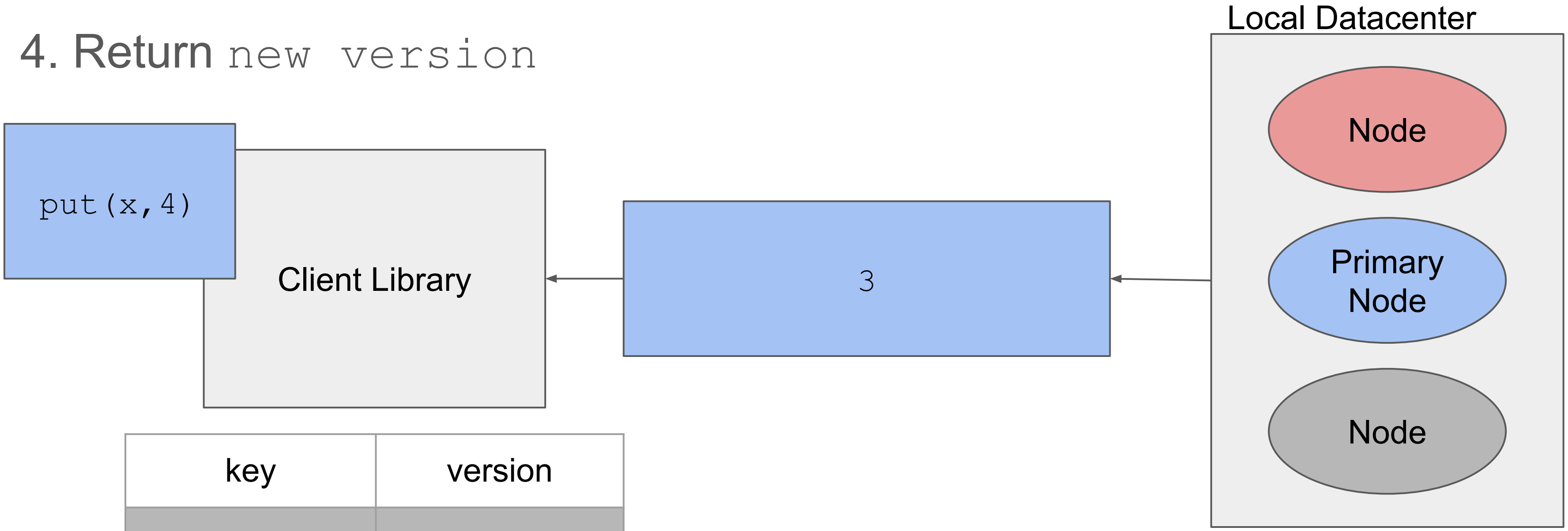
3. Client Library sends `put_after` request



key	version
y	1
z	2

Example `put`

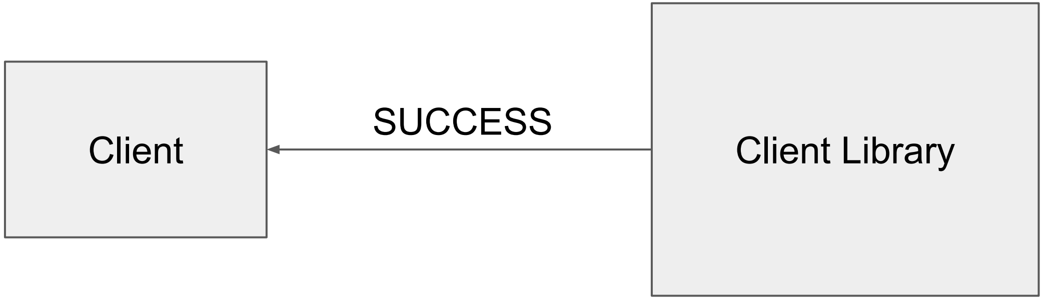
4. Return `new version`



key	version
y	1
z	2

Example `put`

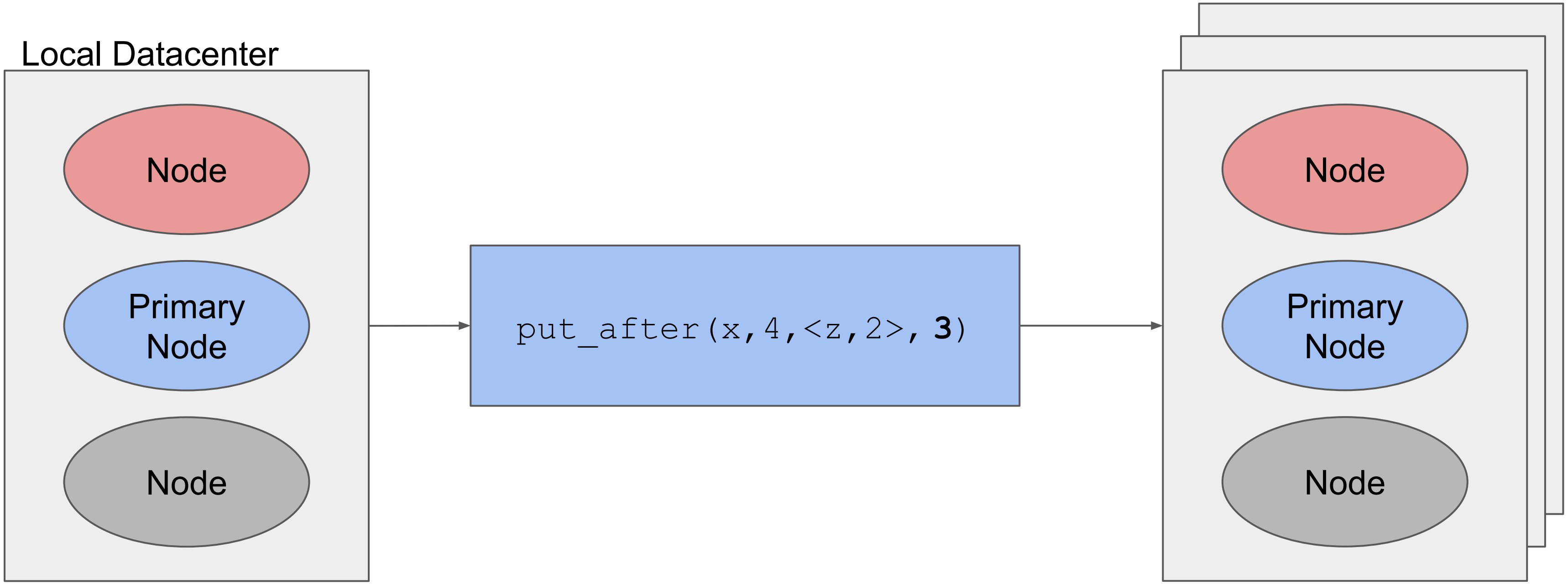
5. Client Library updates metadata and returns



key	version
x	**3**

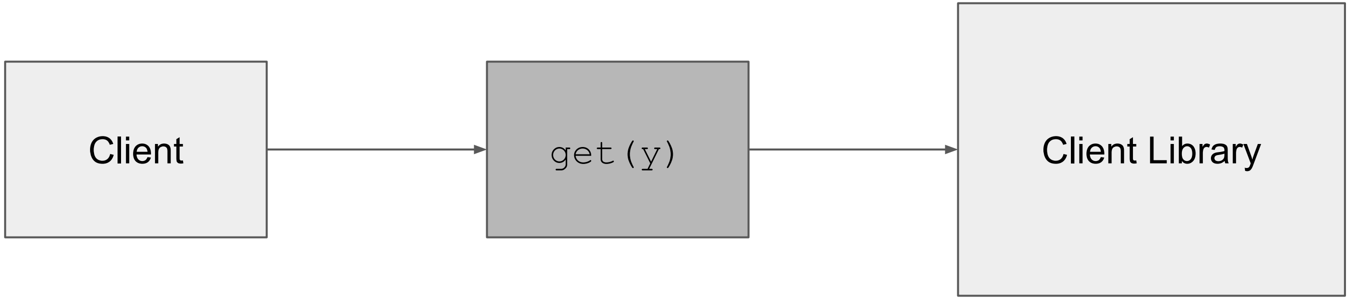
Example `put`

6. Local Datacenter forwards to others



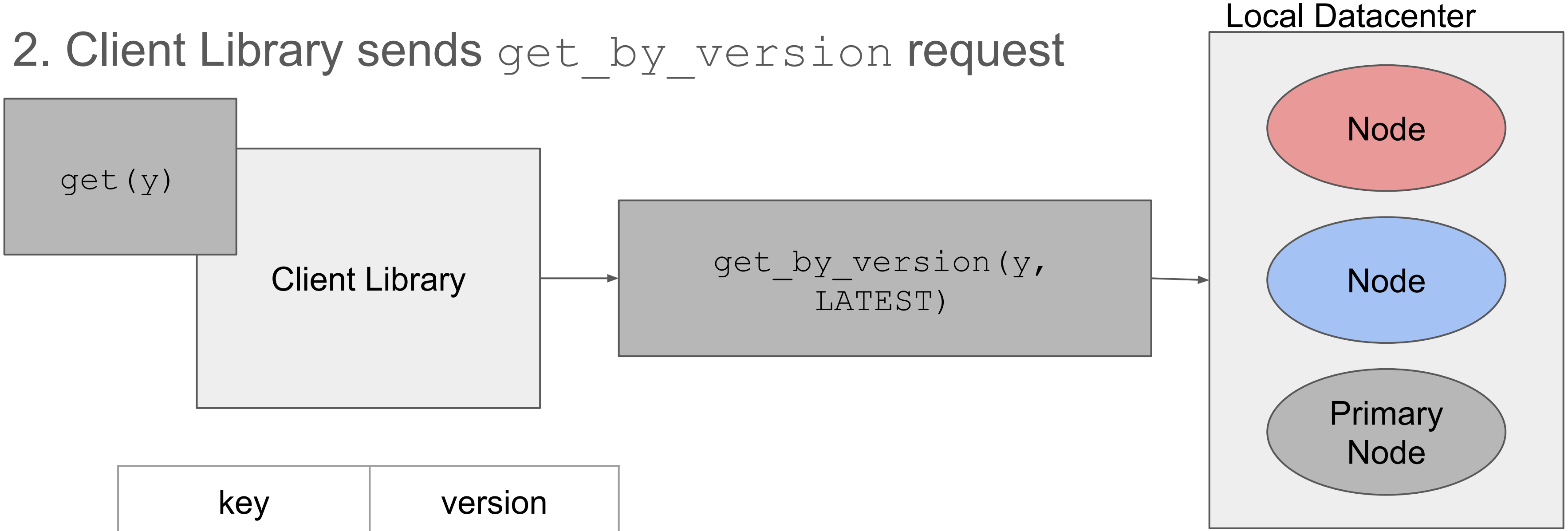
Example `get`

1. Client calls `get(key)`



Example `get`

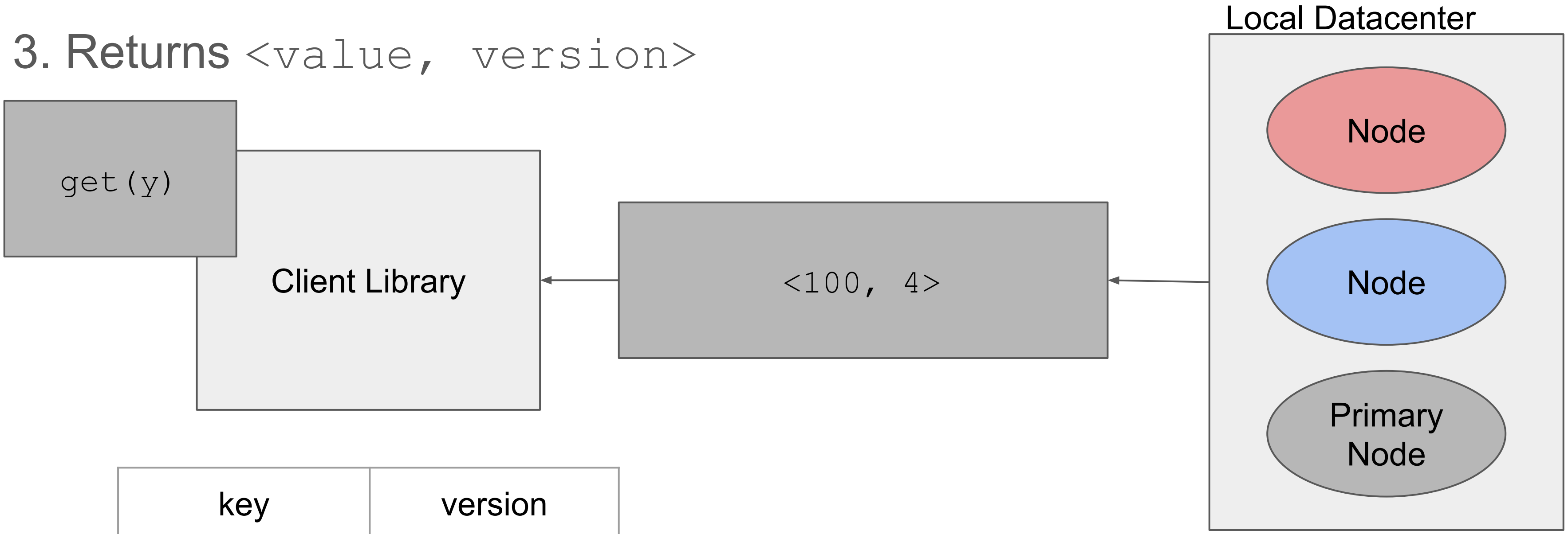
2. Client Library sends `get_by_version` request



key	version
x	3

Example `get`

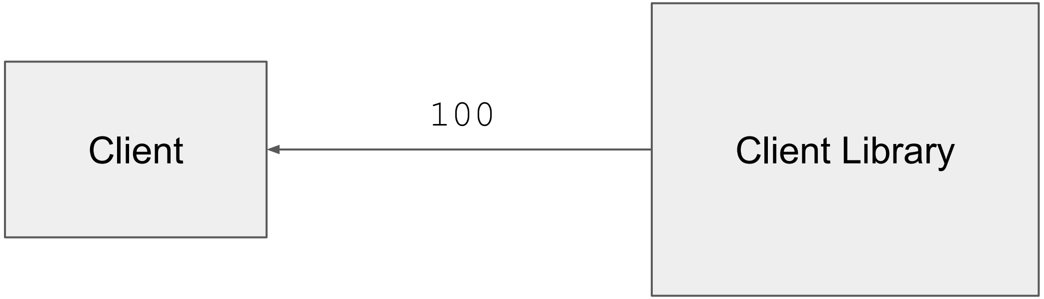
3. Returns `<value, version>`



key	version
x	3

Example `get`

4. Client Library updates metadata and returns



key	version
x	3
y	**4**

We can write values with `put`.

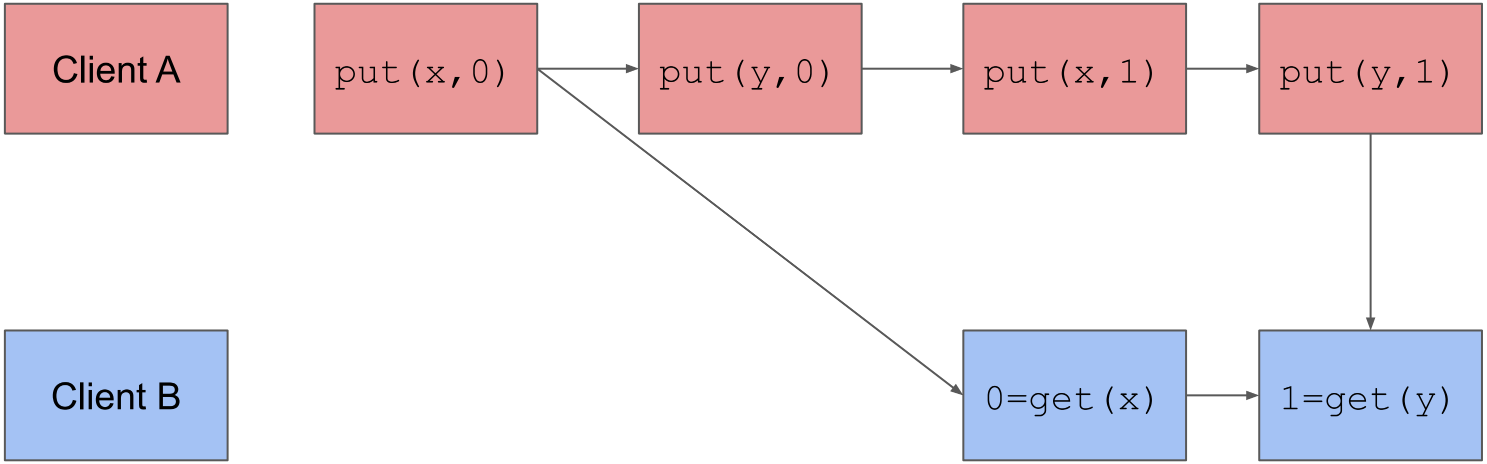
We can retrieve values with `get`.

These operations respect causal+ consistency.

But there is still in issue...

Consistent Dependent `get` Requests

Let `x` and `y` be dependent keys



These values are inconsistent with each other

Client Library Interface

Read a value based on key:

```
val = get(key, ctx)
```

Write a value to a key:

```
put(key, val, ctx)
```

Create context:

```
ctx = createContext()
```

Delete context:

```
bool = deleteContext(ctx)
```

Get collection of keys:

```
<values> = get_trans(<keys>,ctx)
```

COPS-GT Client Library Changes

The client library will be storing `<key, version, dep>` tuples.

On a `get`, retrieved `<key, version, dep>` tuple is added

On a `put`, that key's deps are set to all other keys in that context

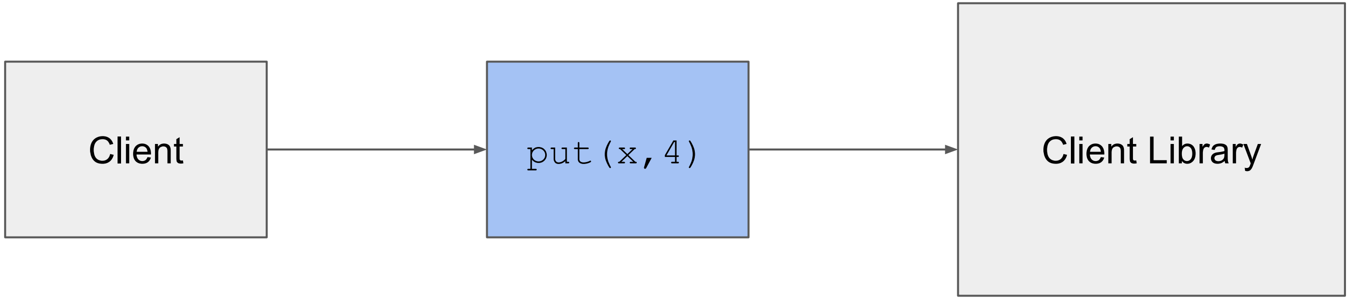
Datacenter Interface Changes

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

<value, version, deps> = get_by_version(key, version)
```

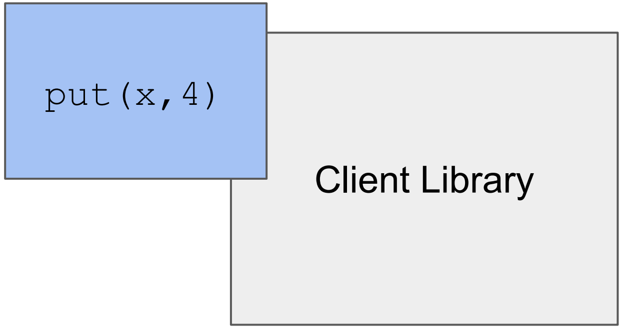
Example `put` (COPS-GT)

1. Client calls `put(key, val)`



Example `put` (COPS-GT)

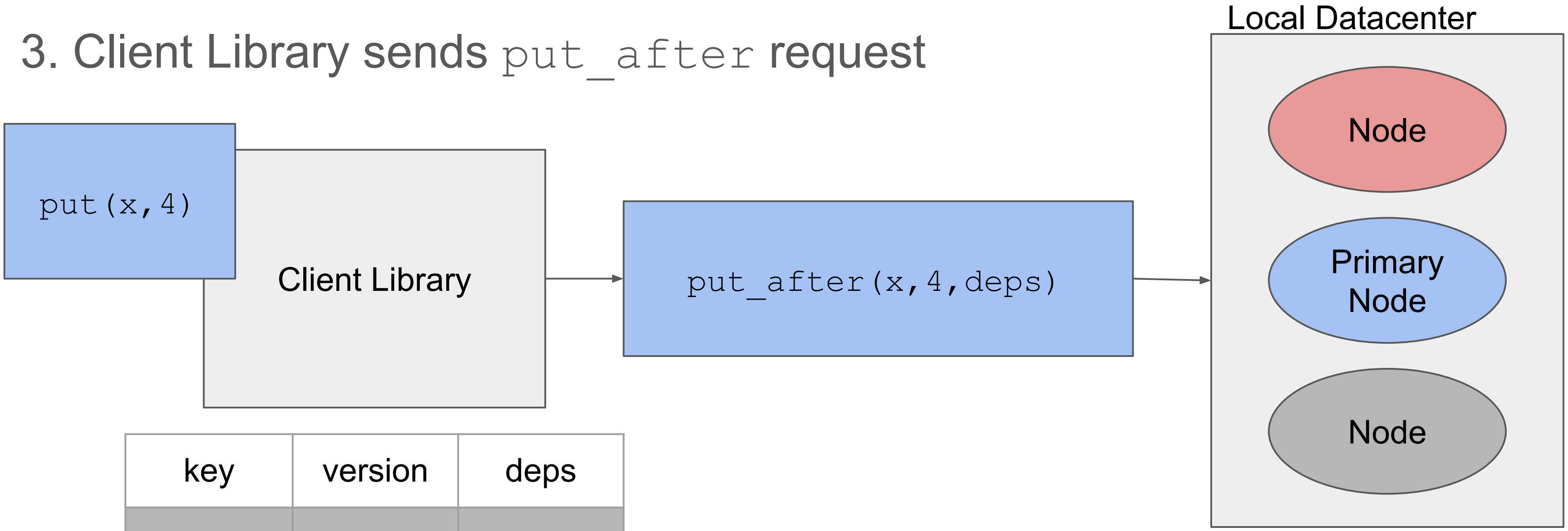
2. Client Library calculates **all** dependencies



key	version	deps
y	1	
z	2	<y, 1>

Example `put` (COPS-GT)

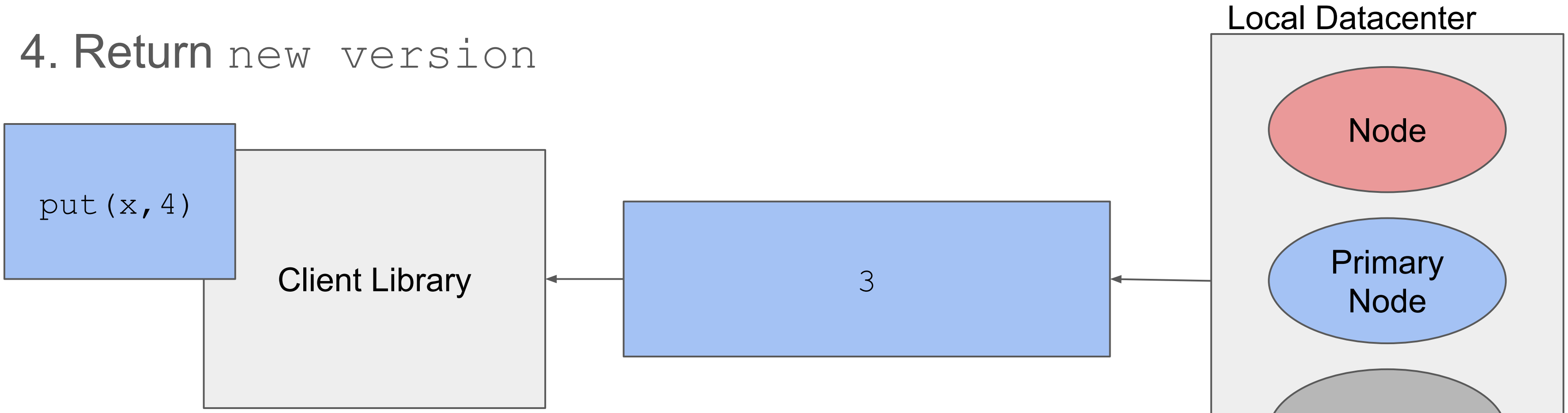
3. Client Library sends `put_after` request



key	version	deps
y	1	
z	2	<y,1>

Example `put` (COPS-GT)

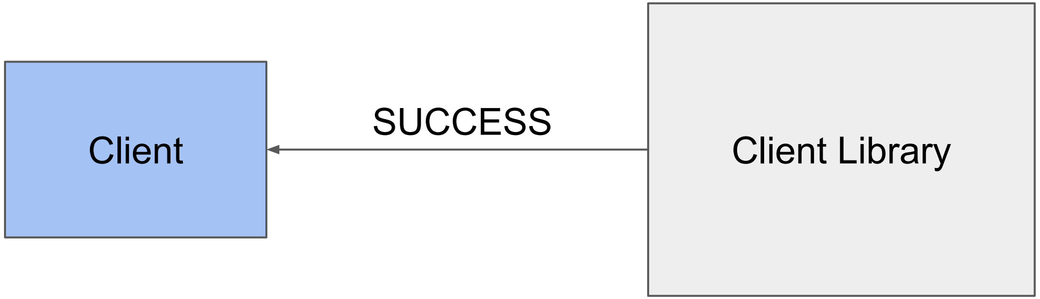
4. Return `new version`



key	version	deps
y	1	
z	2	<y,1>

Example `put` (COPS-GT)

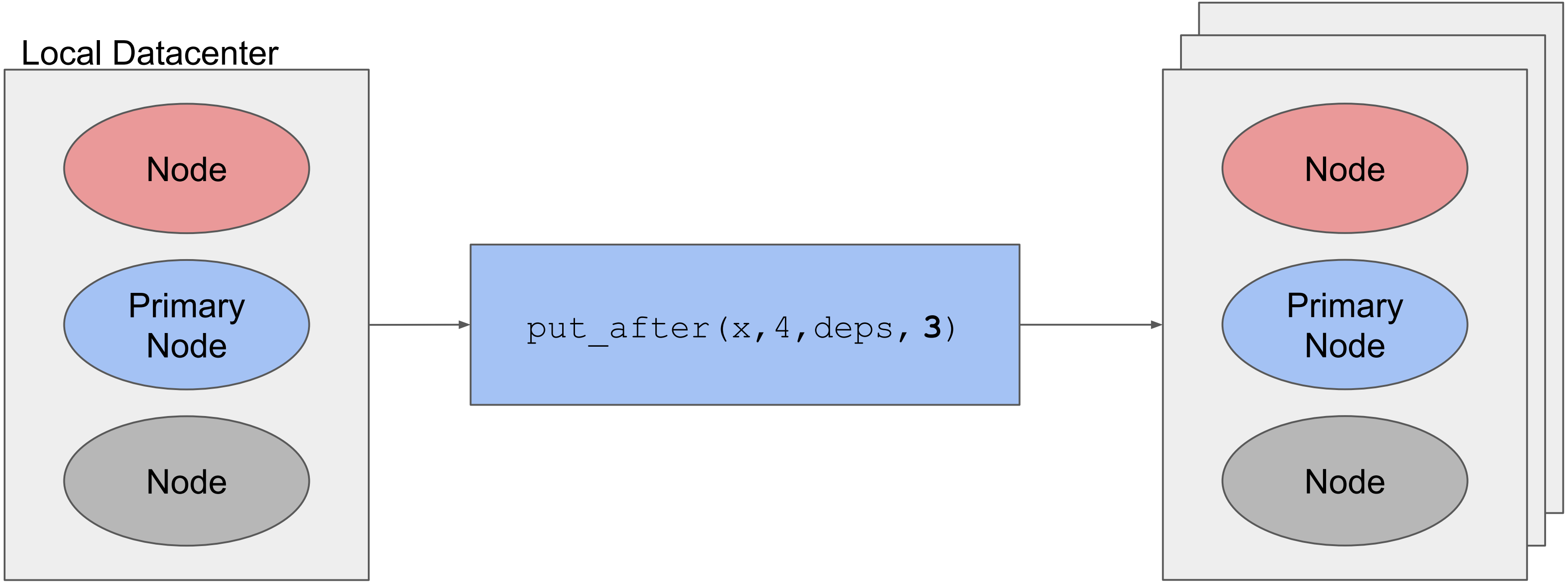
5. Client Library updates metadata and returns



key	version	deps
y	1	
z	2	<y,1>
x	**3**	**<y,1> <z,2>**

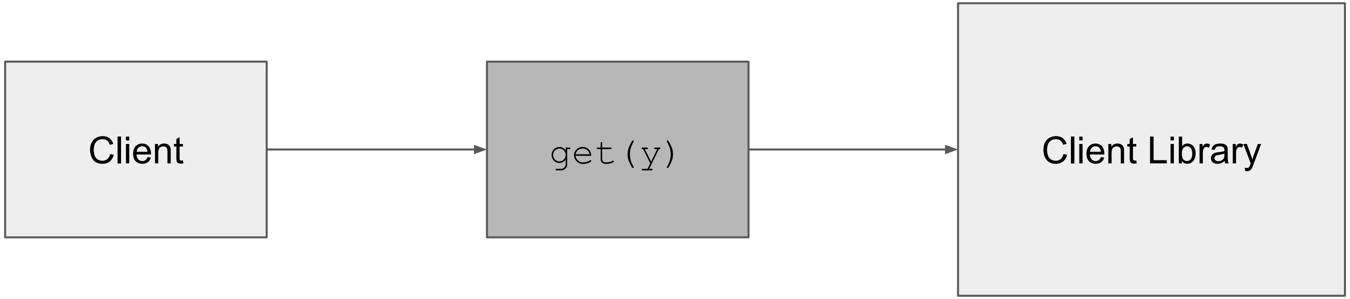
Example `put` (COPS-GT)

6. Local Datacenter forwards to others



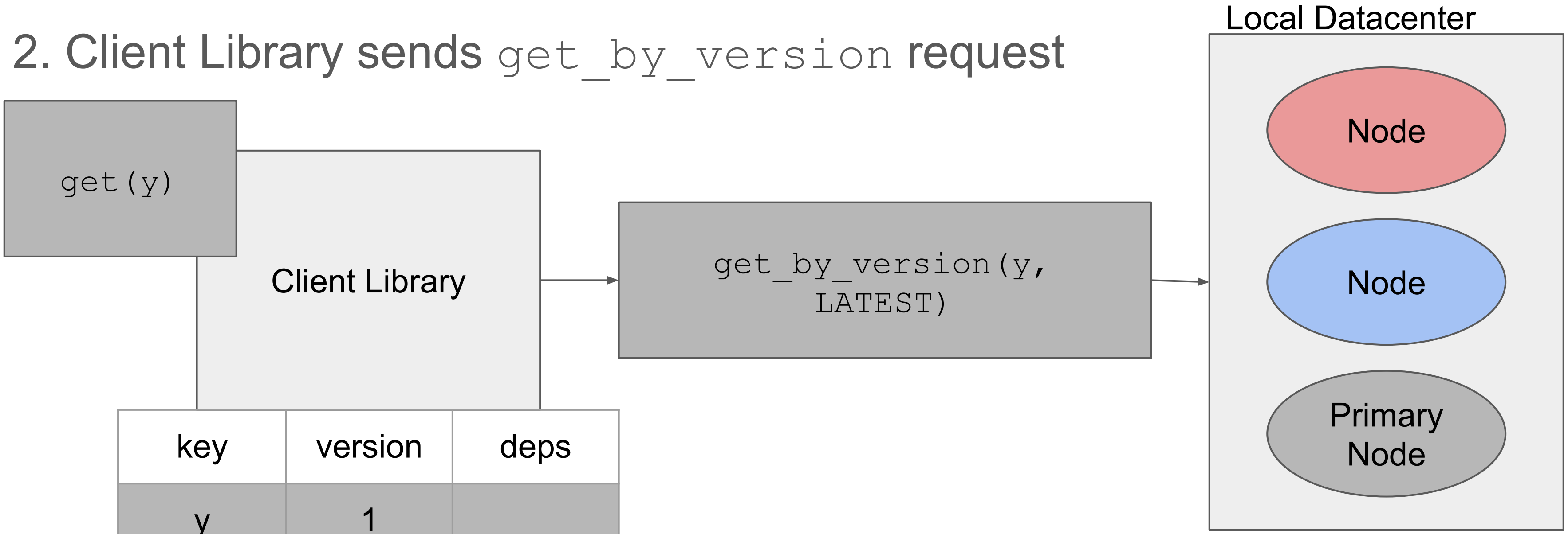
Example `get` (COPS-GT)

1. Client calls `get(key)`



Example `get` (COPS-GT)

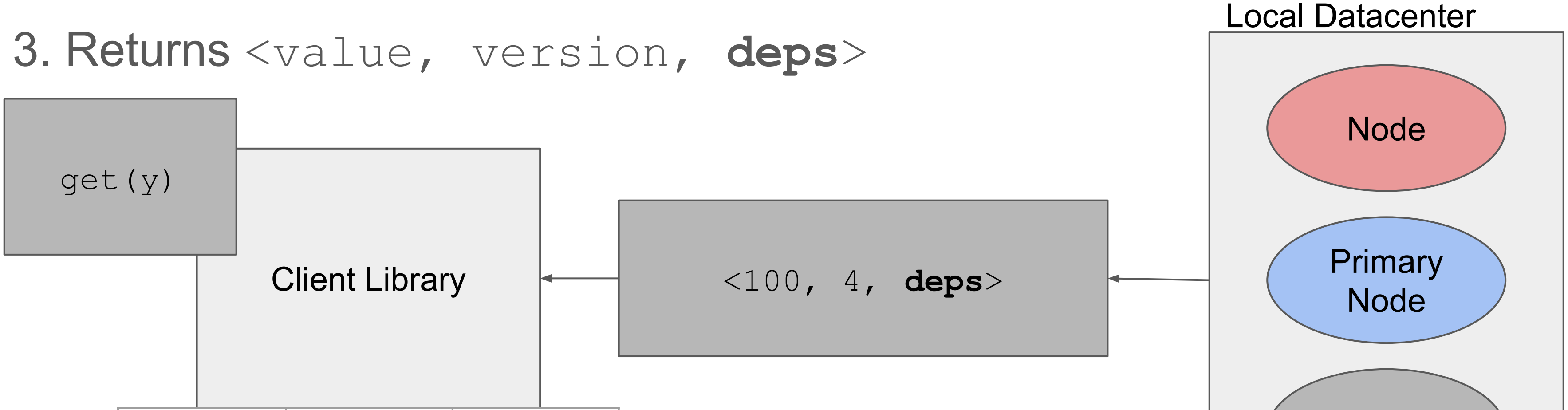
2. Client Library sends `get_by_version` request



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

Example get (COPS-GT)

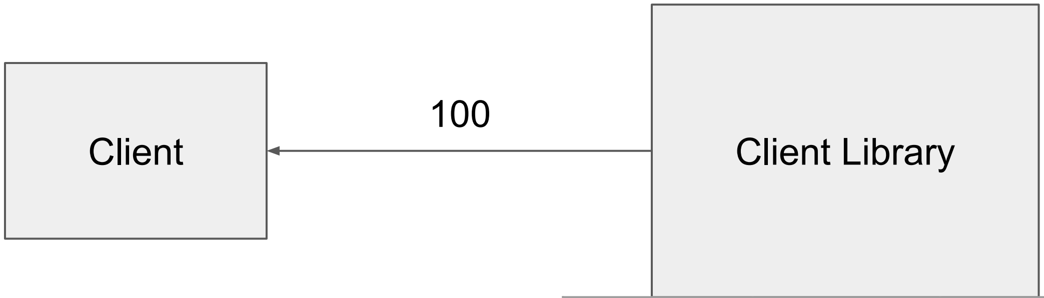
3. Returns <value, version, **deps**>



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

Example `get` (COPS-GT)

4. Client Library updates metadata and returns



key	version	deps
y	**4**	**<z, 2>** **<x, 3>**
z	2	<y,1>
x	3	<y,1> <z,2>

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:

Value Requested	X	Y
Version	2	3
Dependencies	<Z, 5>	<X, 4>
	<Y, 1>	

COPS-GT Get Transaction: Two Rounds

Round 2

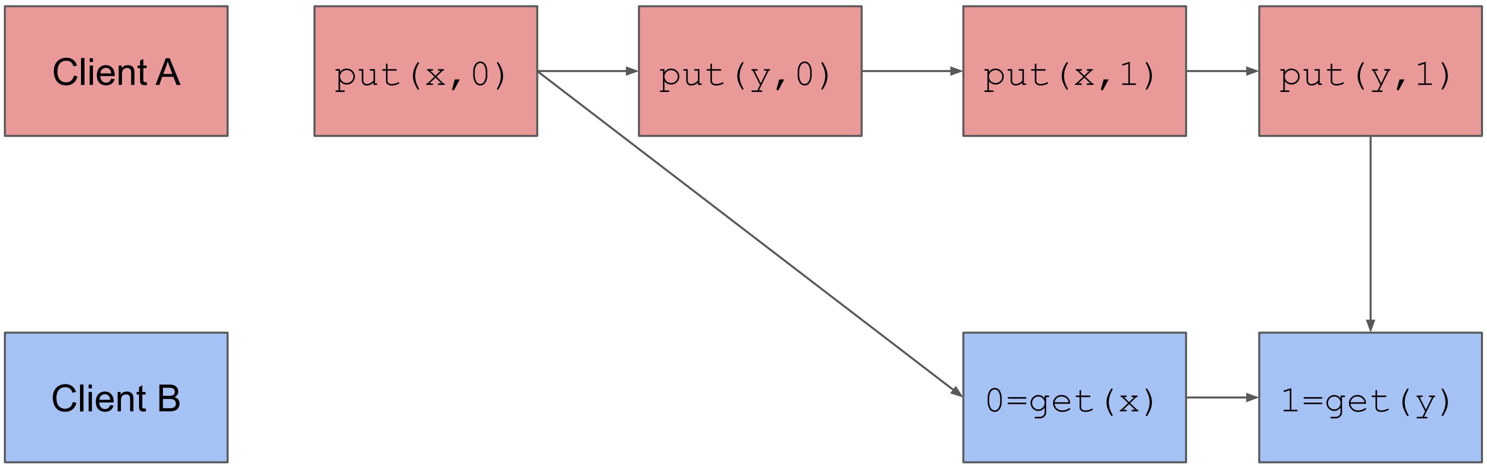
- For each inconsistent key, call `get_by_version` again

In this example, request version 4 of X

Value Requested	X	Y
Version	2	3
Dependencies	<Z, 5>	<X, 4>
	<Y, 1>	

Let's revisit this example

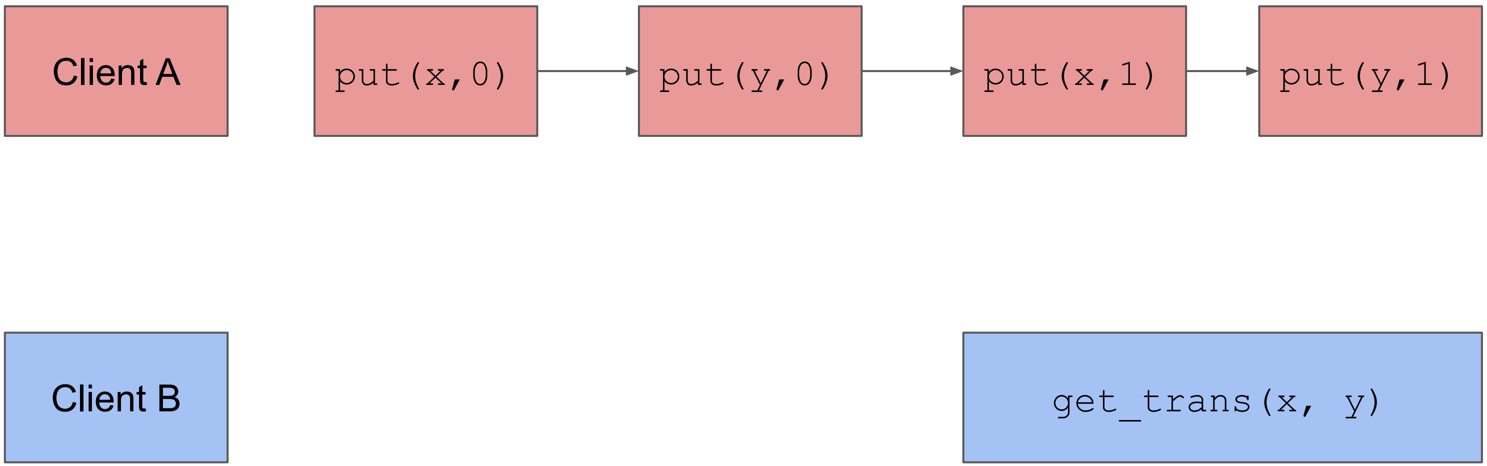
Let `x` and `y` be dependent keys



These values are inconsistent with each other

Let's revisit this example

Let `x` and `y` be dependent keys



Round 1:

Call `get_by_version` for X and Y

Value Requested	X	Y
Version	1	4
Dependencies		<X, 3>

Round 2:

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

Problem Solved!

Garbage Collection

Define a timeout *T* for `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

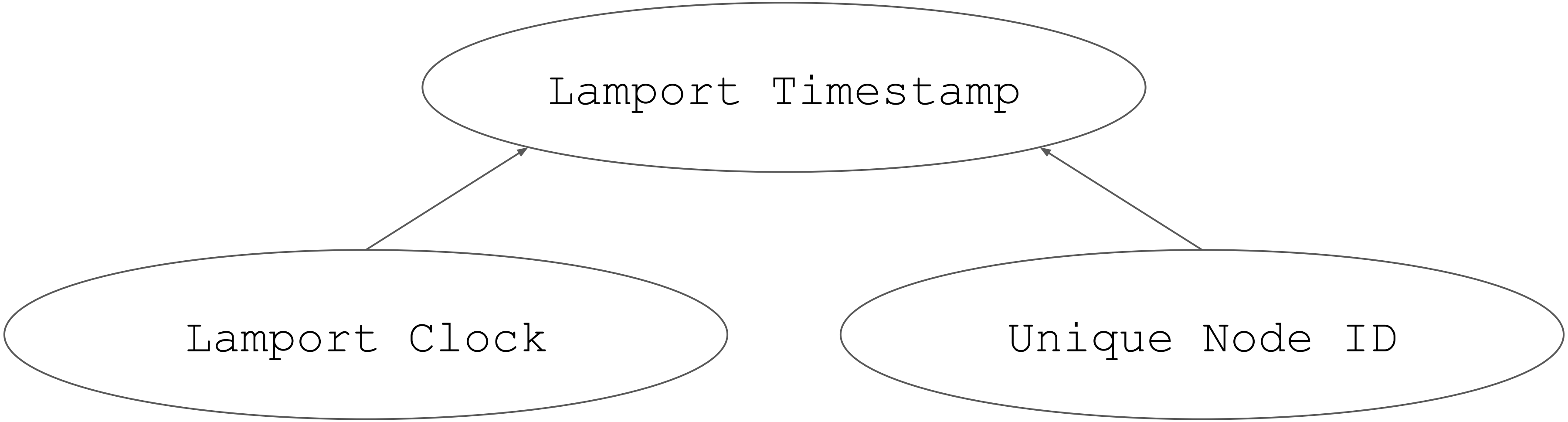
Fault Tolerance

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

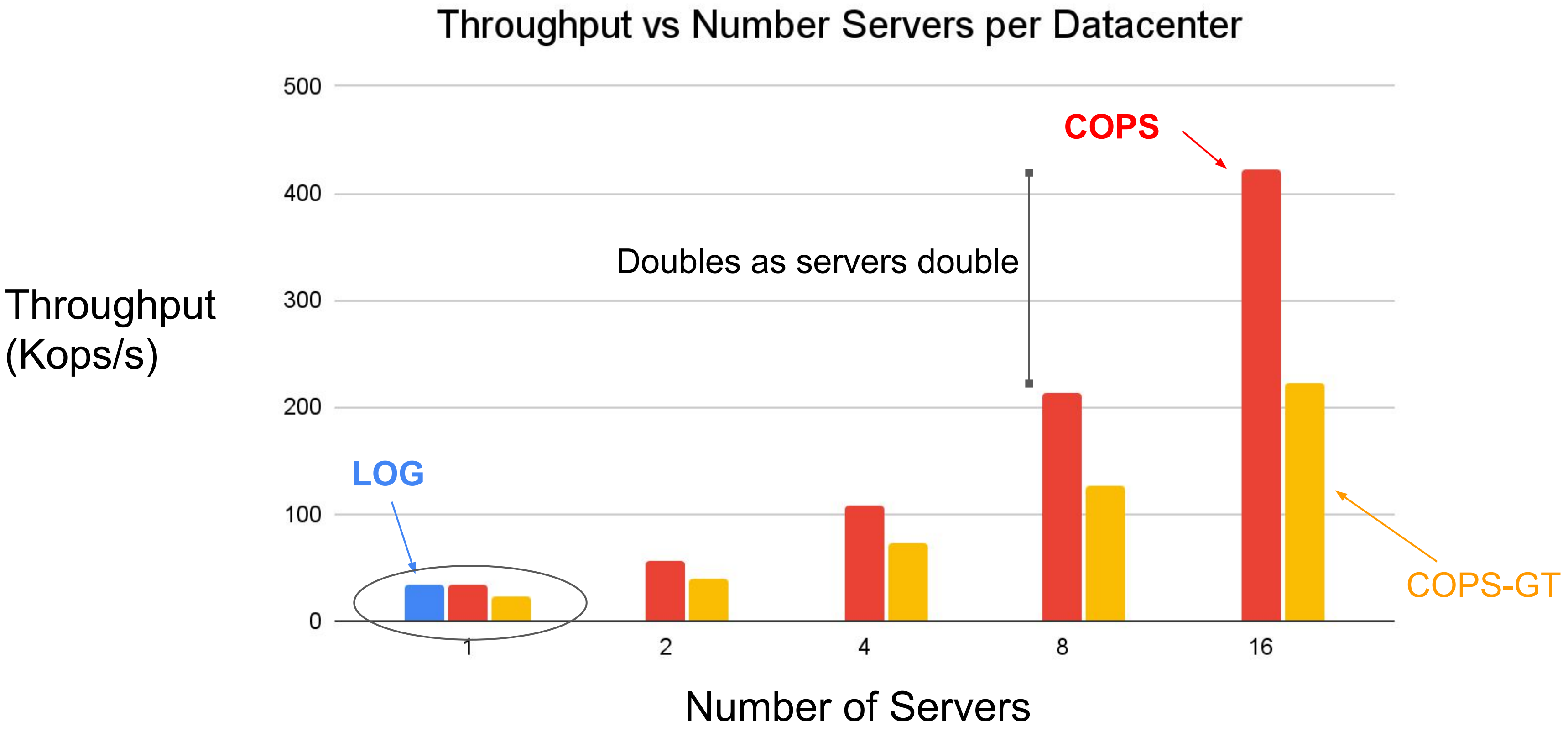
Conflict Detection

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

- Use Lamport Timestamp



Evaluation: Scalability



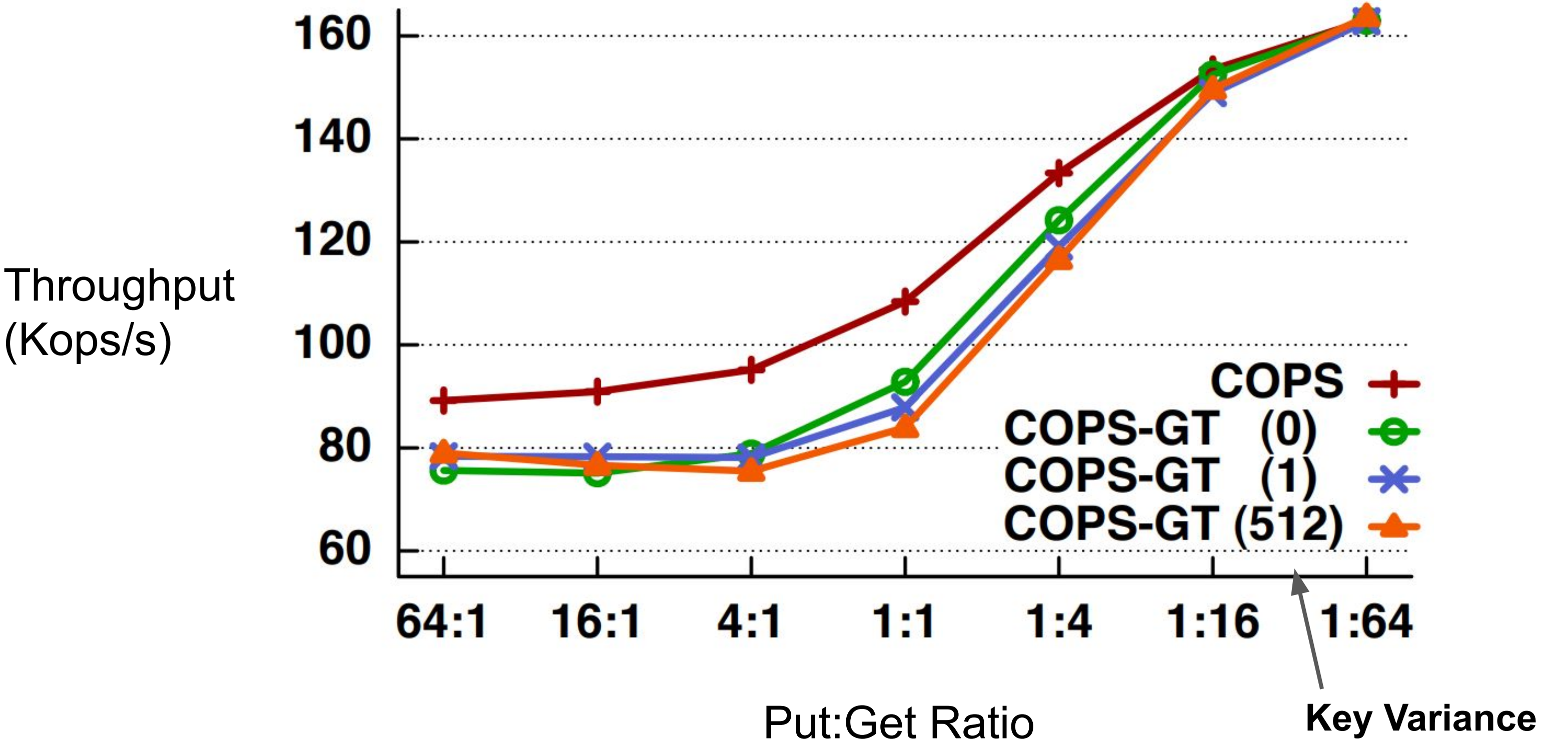
Evaluation: Latency

Control

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

Number of dependencies

Evaluation: Throughput



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