The CAP Theorem Discussion

Presenters: Cuong Pham & Biplap Deka

CS525 Spring 2013

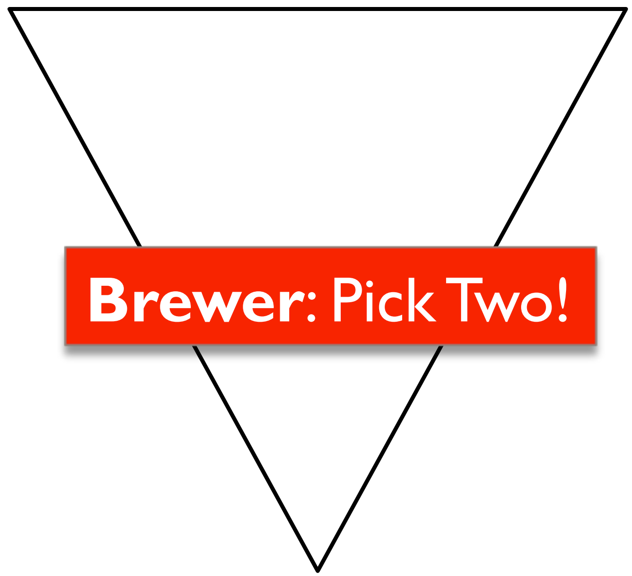
CAP Theorem

Atomic/Linearizable Consistency

Exist a total order of all Operations such that each operation looks as if it were completed at a single instant

Availability

Every request received by a non-failing node must result in a response



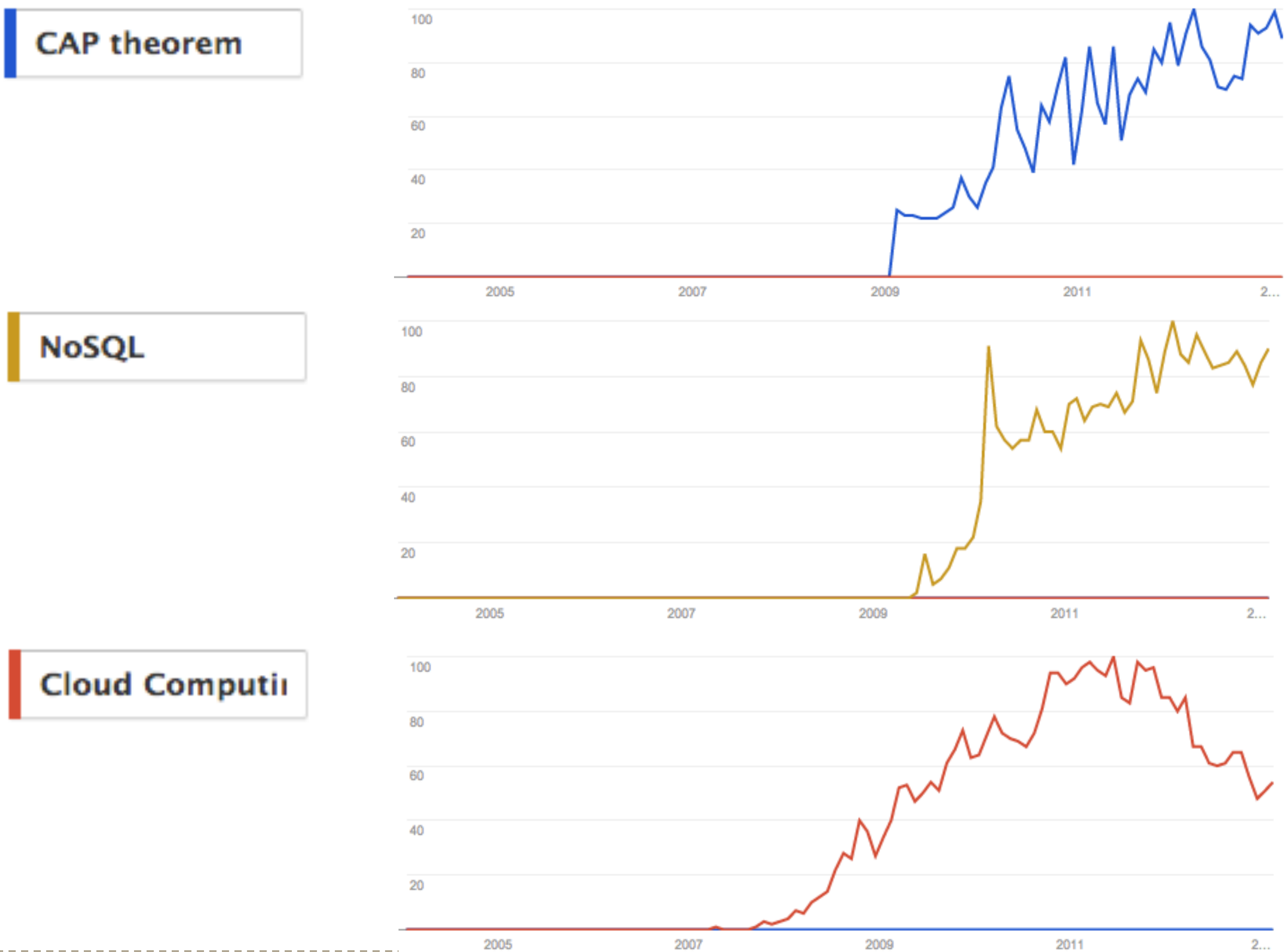
Partition-tolerance

No set of failures less than total network failure Is allowed to cause the system to response incorrectly

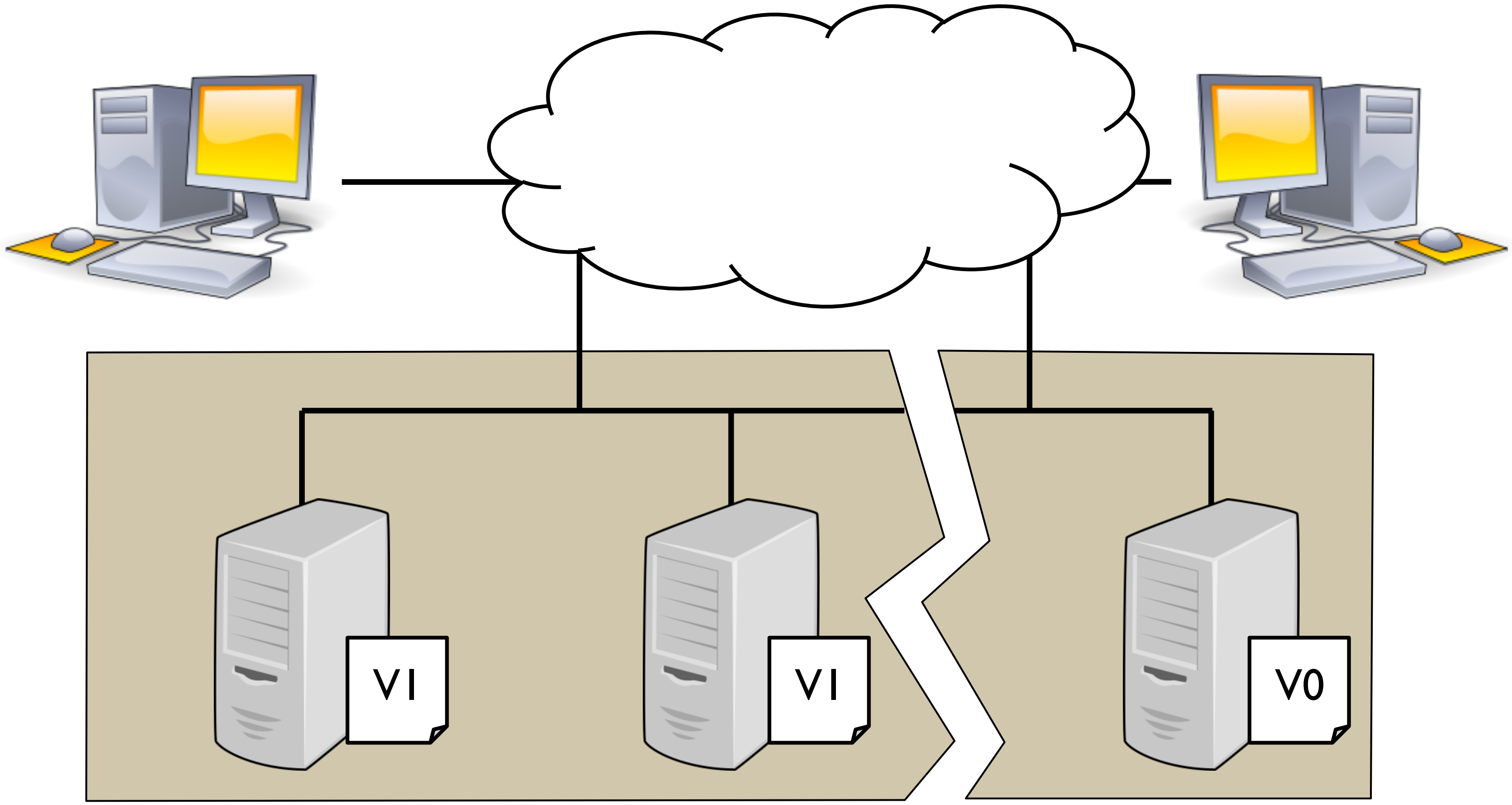
Historical Context

- **2000** — E. Brewer stated CAP trade-offs in his keynote talk at PODC
- **2002** — S. Gilbert & N. Lynch published a formal proof of CAP
- **2009** — NoSQL movement began
- **2012** — S. Gilbert and N. Lynch: "*Perspective on the CAP theorem*"
  E. Brewer: "*CAP Twelve Years Later: How the "Rules" Have Changed*"
- **Today** — 150 NoSQL Databases (nosql-database.org)

CAP, Cloud Computing, and NoSQL in Google Trends



Proof Sketch



Consistency is Important

- Affect the correctness of data
- People are used to its strong notion in pre-cloud era
- The ONLY knob you can tune in CAP in many scenarios

- **RedBlue Consistency**
  - C. Li, el at., “Making geo-replicated systems fast as possible, consistent when necessary,” In OSDI, Oct 2012.

- **Causal+ Consistency**
  - W. Lloyd, el at., "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS," In SOSP 2011.

CAP Theorem Debate

- With the wide geographical spread of the *cloud*, opportunities for partitioning in the data are not in-significant
  - An justification for NoSQL Movement
- Latency is not in the equation (e.g. PNUT)
- A and C are asymmetric
- Differences between CA and CP?

Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary

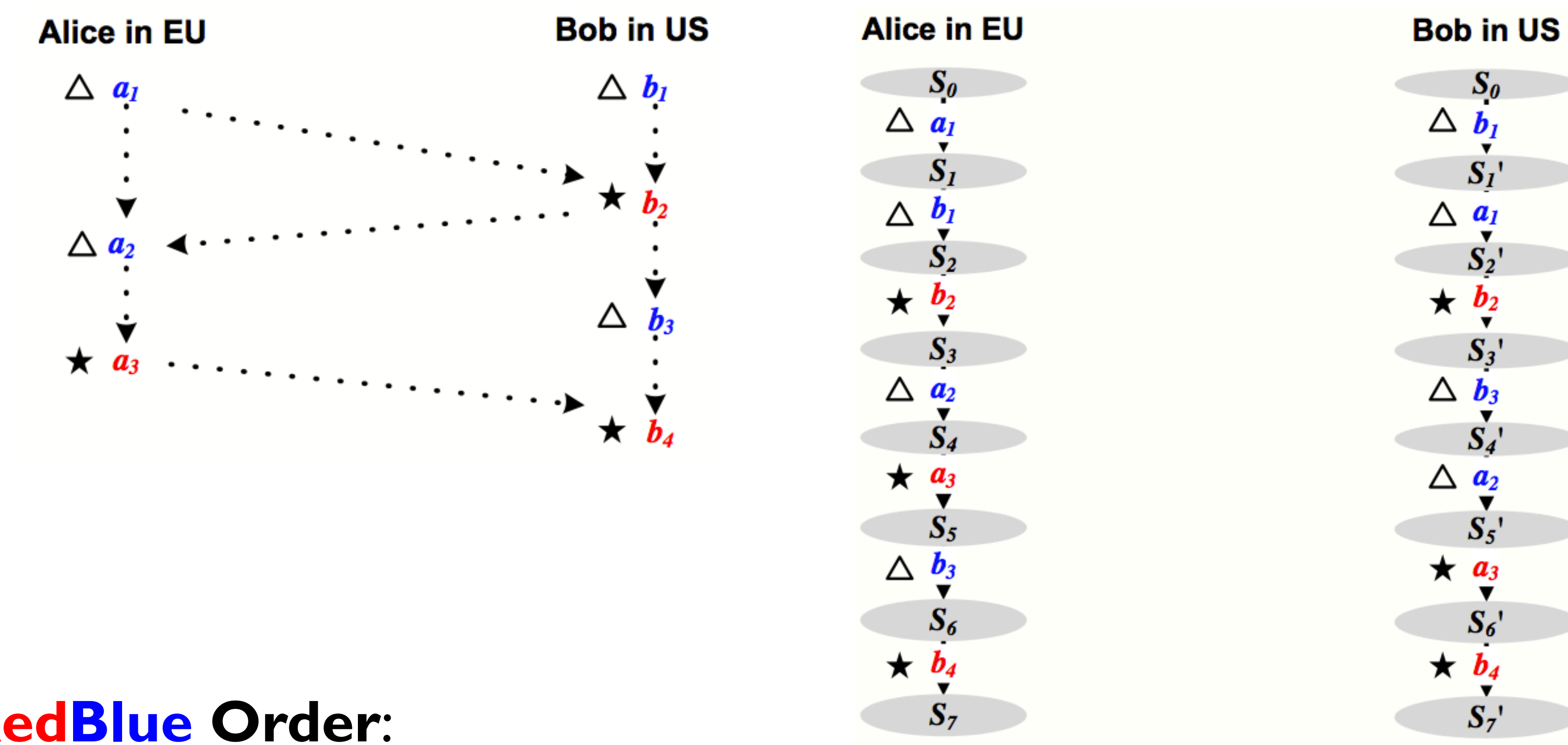
Authors: Chen Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguica, Rodrigo Rodrigues

Presenter: Cuong Pham

Motivation

- Geo-replication
  - Internet users are globally distributed
  - Applications replicate data across datacenters
    - Reduce network latencies to users
  - Dilemma:
    - Cross-site consistency latency
    - The problems are magnified with WAN latency
    - E.g.: Synchronous replication via Paxos.
- Observation:
  - Strong consistency is not always required: Depend on the applications
- Goal:
  - **RedBlue Consistency**: Mixing strong consistency (for application semantics) & eventual consistency (for fast responses) in a same system

Divide Operations into Red and Blue

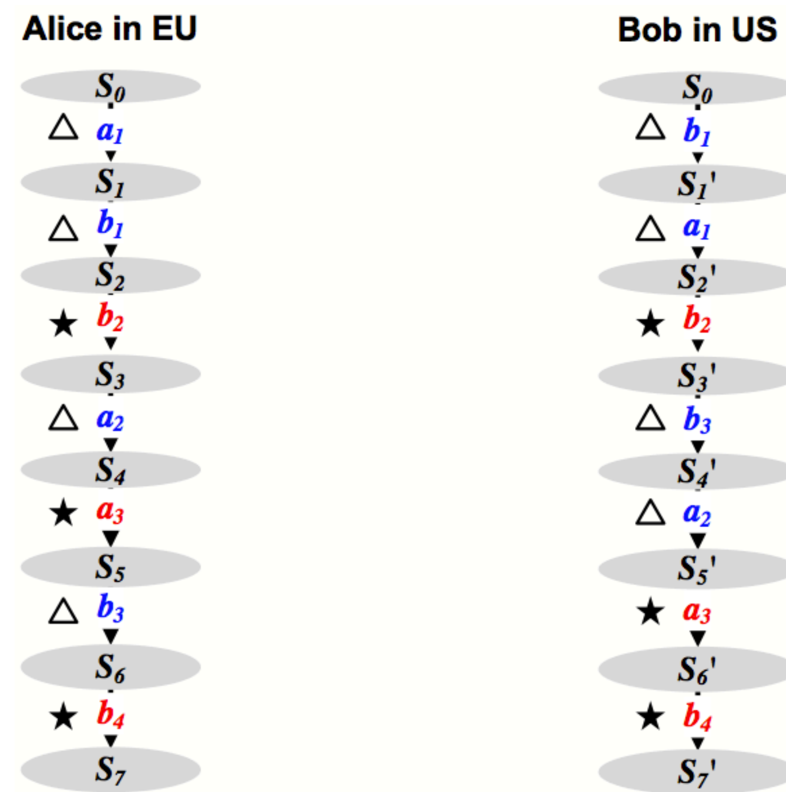


RedBlue Order:

- Red operations must be totally ordered
- The order of Blue operations can vary from site to site

RedBlue Consistency

- **Causal serialization**
  - A site has a causal serialization of the RedBlue order if the ordering is *a linear extension of the RedBlue order*
- **State Convergence**
  - Convergent if all causal serializations of the RedBlue order *reach the same state*
  - All *blue operations must be globally commutative*



RedBlue Consistency: Each site applies operations according to the causal serialization of the RedBlue order

RedBlue Consistent Banking System

Alice in EU	**Bob in US**
△ *deposit(20)*	△ *accrueinterest()*

(a) RedBlue order O of operations issued by Alice and Bob

```
deposit(float money) {
  balance = balance + money;
}

withdraw(float money) {
 if ( balance - money >= 0 )
   balance = balance − money;
 else
   print "failure";
}

accrueinterest() {
  float delta = balance ×
interest;
  balance = balance + delta;
}
```

Problem:

- Different execution orders lead to divergent state

Cause:

- *Accrueinterest()* doesn't commute with *deposit()*

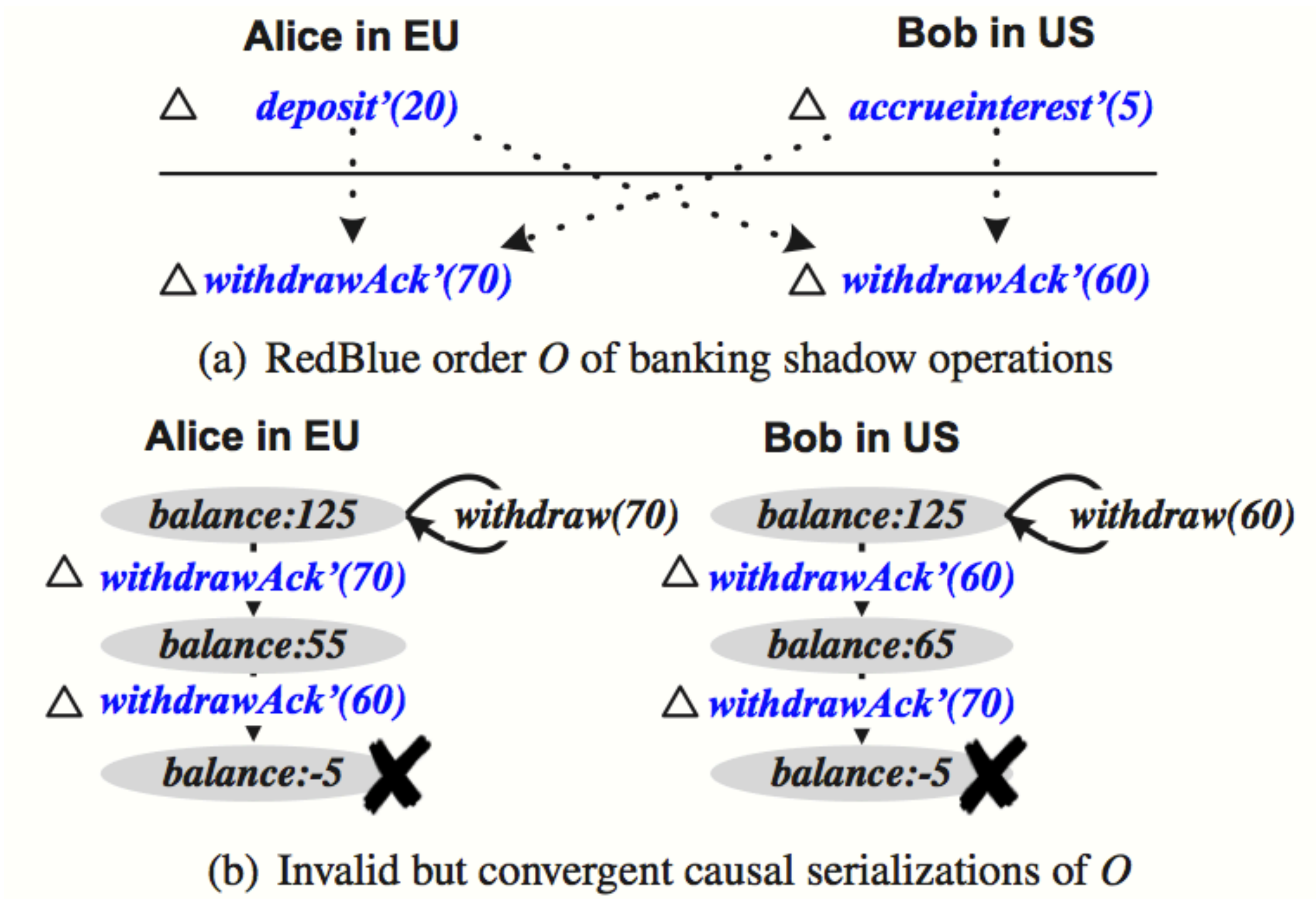
Operation Decomposition
Generator & Shadow operations

- Observation: Not all operations are commutative
- Split these operations into **generator** and **shadow** operations
- Generator Operations
  - Only executed at the primary site against a system state
  - Produces no side effects
  - Determines state transitions that would occur
  - Produces shadow operations
- Shadow Operations
  - Applies the state transitions to all the sites including the primary site
  - Must produce the same effects as the original operation given the original state for the Generator operation
- Separating operations allows for easier formation of abelian groups
  - Allows for more commutative operations (blue operations)

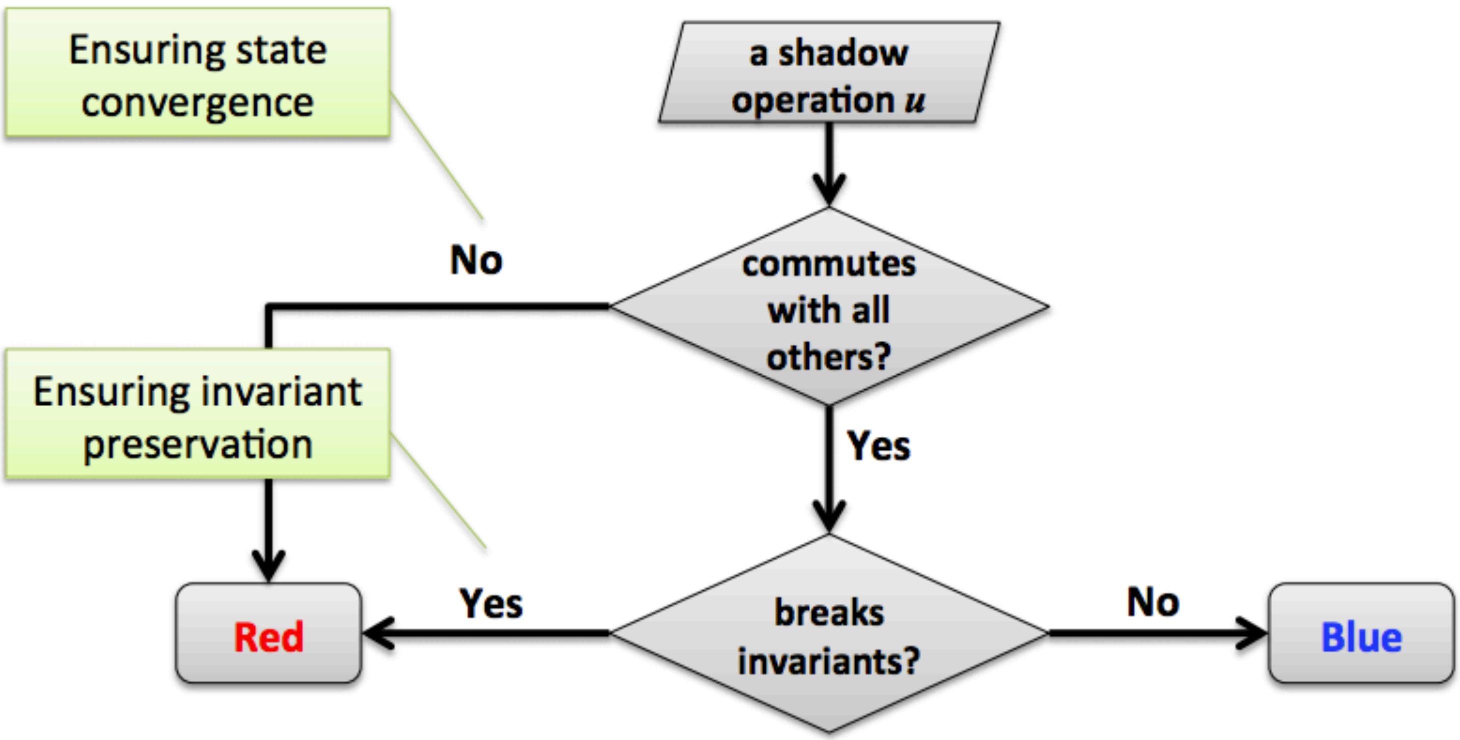
Revisit the Banking System

Original/Generator operation		Shadow operation
deposit(**float** *m*){ *balance = balance + m;* }	*produces*	deposit'(**float** *m*){ *balance = balance + m;* }
accrueinterest(){ **float** *delta=balance × interest; balance=balance + delta;* }	*produces*	accrueinterest'(**float** *delta*){ *balance=balance + delta;* }
withdraw(**float** *m*){ **if**(*balance-m>=0*) *balance=balance - m;*	*produces*	withdrawAck'(**float** *m*) { *balance=balance - m;* }
else *print "Error"* }	*produces*	withdrawFail'(){ }

Converged… but Invalid

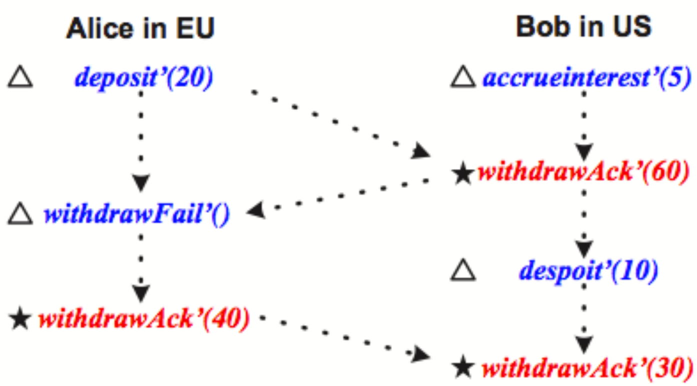


Red or Blue?

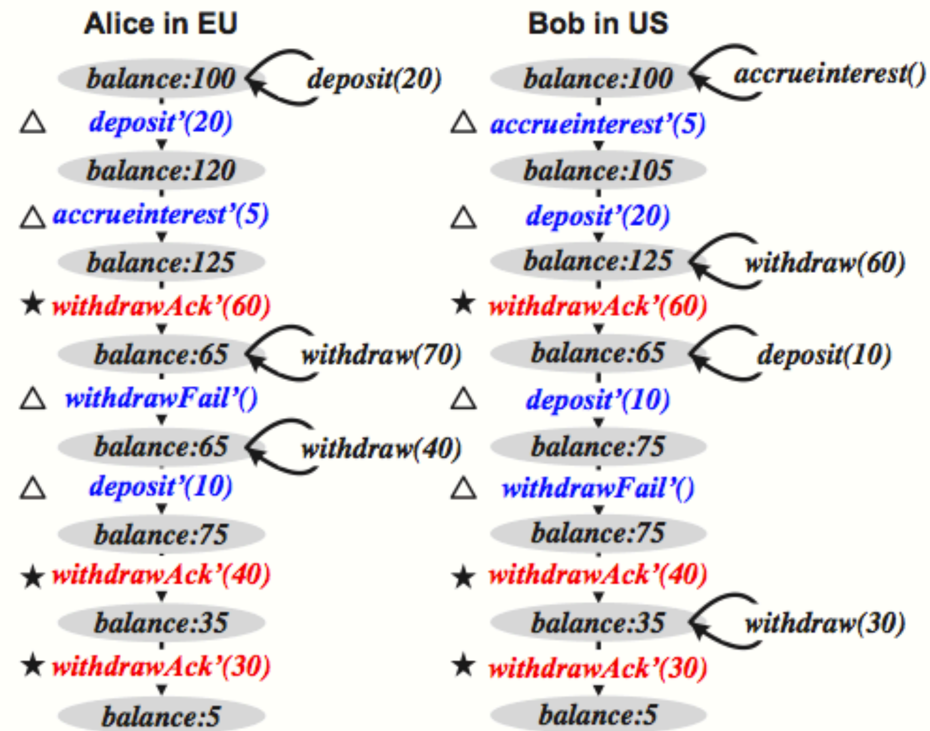


Credit: Authors' slides

Correct RedBlue Consistent Banking



(a) RedBlue order O of banking shadow operations



(b) Convergent and invariant preserving causal serializations of O

Summary

- RedBlue consistency combines strong and eventual consistency into a single system
- The decompositon of generator/shadow operations expands the space of possible Blue operations
- A simple rule for labeling is provably state convergent and invariant preserving

Evaluations

Experimental Setup

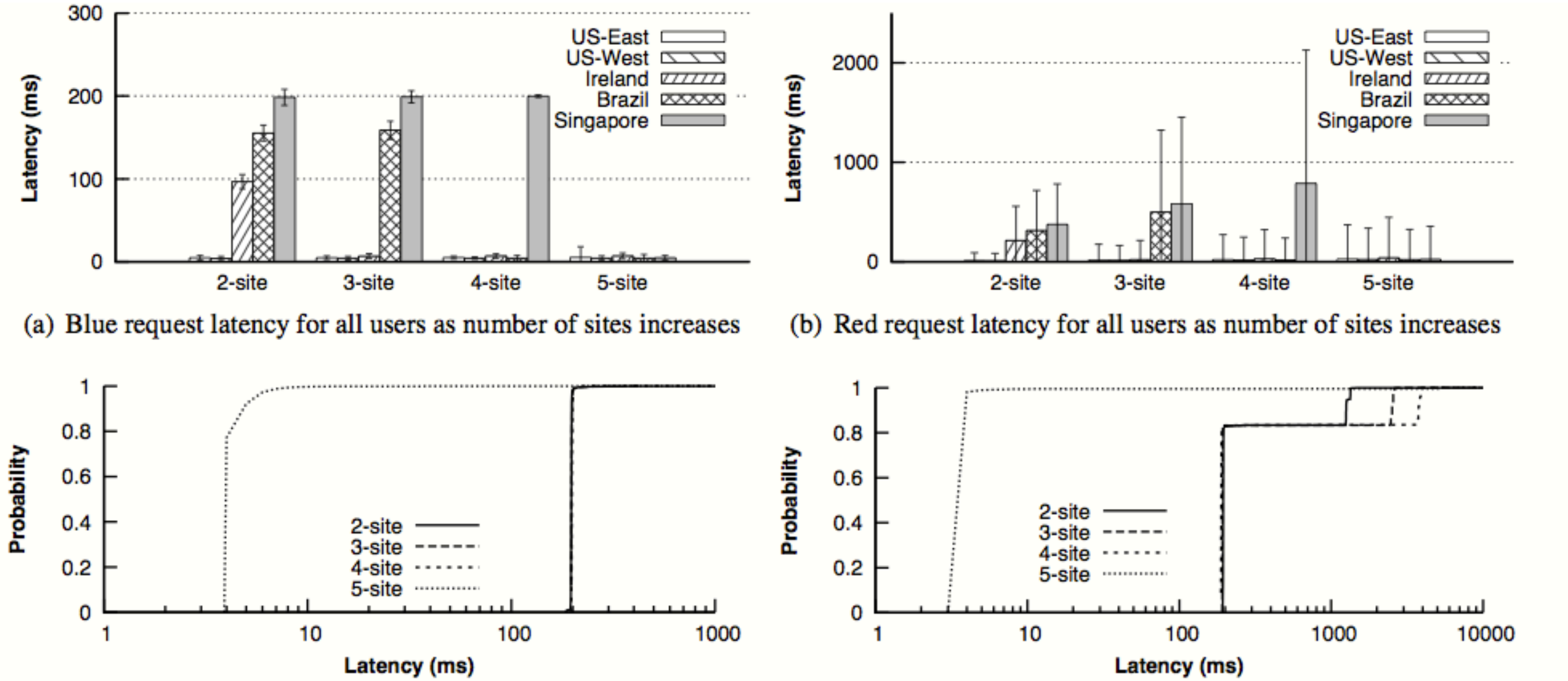
- Deployment in Amazon EC2
  - spanning 5 sites (US-East, US-West, Ireland, Brazil, Singapore)
- Locating users in all five sites and directing their requests to closest server

	UE	UW	IE	BR	SG
UE	0.4 ms 994 Mbps	85 ms 164 Mbps	92 ms 242 Mbps	150 ms 53 Mbps	252 ms 86 Mbps
UW		0.3 ms 975 Mbps	155 ms 84 Mbps	207 ms 35 Mbps	181 ms 126 Mbps
IE			0.4 ms 996 Mbps	235 ms 54 Mbps	350 ms 52 Mbps
BR				0.3 ms 993 Mbps	380 ms 65 Mbps
SG					0.3 ms 993 Mbps

Table 3: Average round trip latency and bandwidth between Amazon sites.

Micro-benchmark Results

Avoid the cost of cross-site communication as much as possible



(a) Blue request latency for all users as number of sites increases (b) Red request latency for all users as number of sites increases

(c) Blue latency CDF for Singapore users as number of sites increases (d) Red latency CDF for Singapore users as number of sites increases

Figure 8: (a) and (b) show the average latency and standard deviation for blue and red requests issued by users in different locales as the number of sites is increased, respectively. (c) and (d) show the CDF of latencies for blue and red requests issued by users in Singapore as the number of sites is increased, respectively.

Case Studies

- Applications
  - Two e-commerce benchmarks: TPC-W, RUBiS
  - One social networking app: Quoddy
- How common Blue operations are?

Application	Original					RedBlue consistent extension				
	user requests	transactions			LOC	shadow operations				LOC changed
		total	read-only	update		blue no-op	blue update	red	LOC	
TPC-W	14	20	13	7	9k	13	14	2	2.8k	429
RUBiS	26	16	11	5	9.4k	11	7	2	1k	180
Quoddy	13	15	11	4	15.5k	11	4	0	495	251

Table 2: Original applications and the changes needed to make them RedBlue consistent.

How common Blue operations are?

- Runtime Blue/Red ratio in different applications with different workloads:

Apps	workload	Originally		With shadow ops	
		Blue (%)	Red(%)	Blue (%)	Red(%)
TPC-W	Browsing mix	96.0	4.0	99.5	0.5
	Shopping mix	85.0	15.0	99.2	0.8
	Ordering mix	63.0	37.0	93.6	6.4
RUBiS	Bidding mix	85.0	15.0	97.4	2.6
Quoddy	a mix with 15% update	85.0	15.0	100	0

Scalability Evaluation

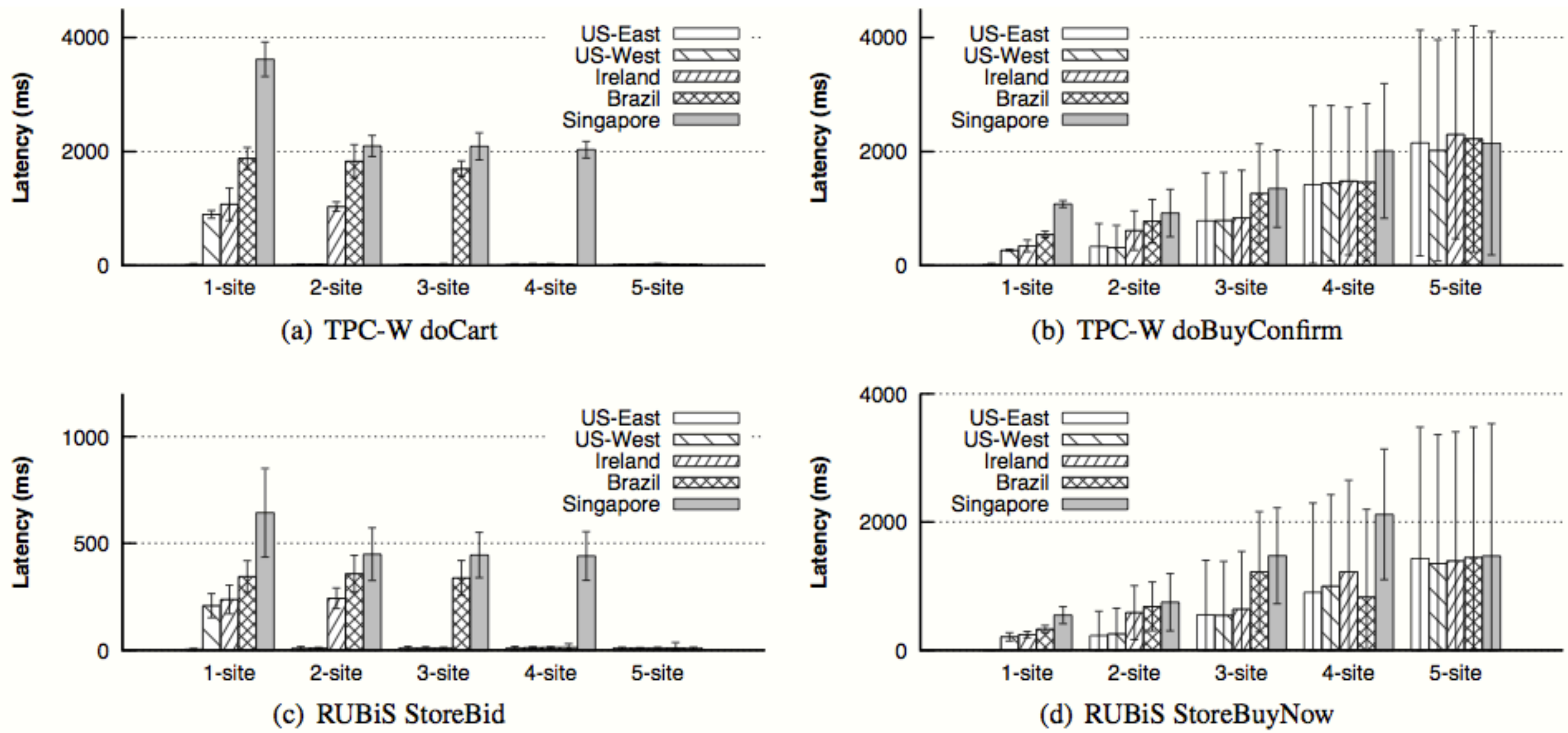


Figure 10: Average latency for selected TPC-W and RUBiS user interactions. Shadow operations for doCart and StoreBid are always blue; for doBuyConfirm and StoreBuyNow they are red 98% and 99% of the time respectively.

Discussion

- Total order in one site: Red operations block Blue operations?
- Operation decomposition:
  - A manual process (how to automate?)
  - Error-prone
- Compare with Cassandra's three consistent levels (One, Quorum, and All)
- How improvements in network technology (e.g. cross-site latency is a few ms) impact future designs of geo-rep. system?
- The idea of RedBlue operation is similar to:
  - Generalized Paxos
  - Generic Broadcast
- Gemini implementation
  - No Fault-tolerance
  - Bottleneck: serialize Red operations via token passing

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

Wyatt Lloyd[*], Michael J. Freedman[*], Michael Kaminsky[†], David G. Andersen[‡]

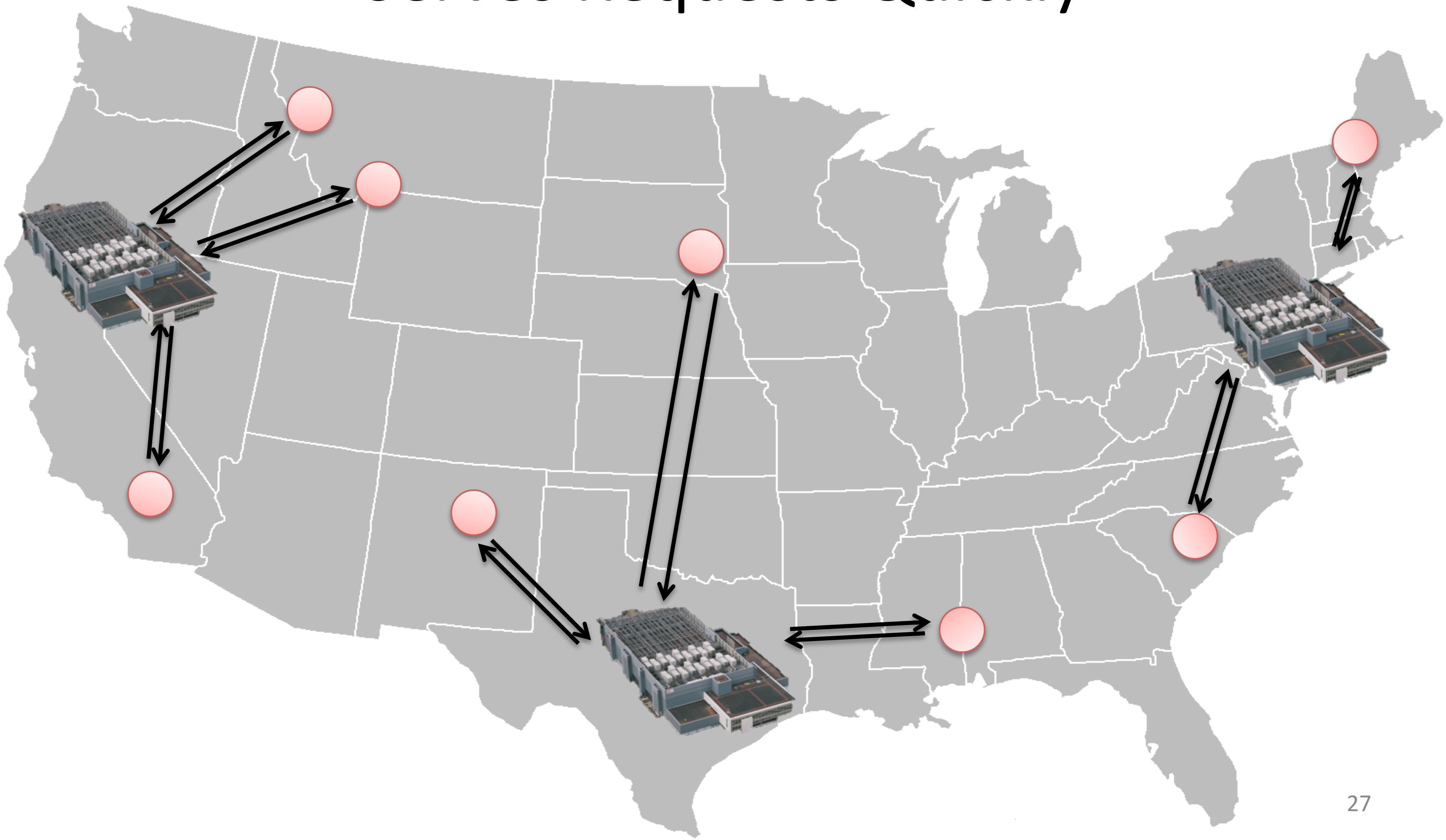
[*]Princeton, [†]Intel Labs, [‡]CMU

SOSP 2011

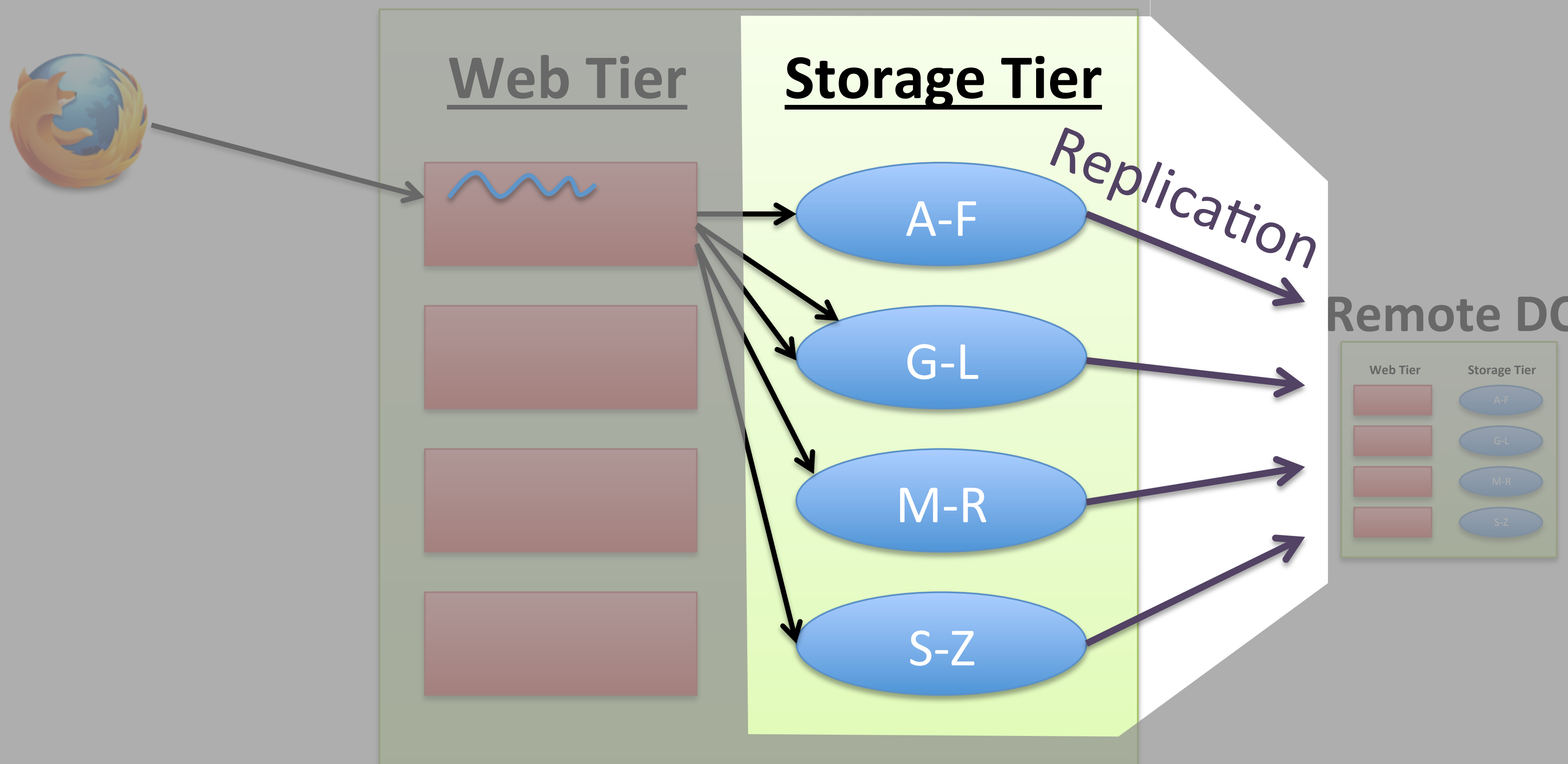
CS525 Presenter : Biplab Deka

Disclaimer: Several slides are borrowed from the presentation by the first author at SOSP 2011

Wide-Area Storage
Serves Requests Quickly



Inside the Datacenter



Desired Properties: ALPS

- **A**vailability
  - All operations to the datastore complete
- **L**ow Latency
  - Client operations complete quickly
- **P**artition Tolerance
  - The datastore continues to work under network partitions
- **S**calability
  - The datastore scales out linearly

"Always On"

Consistency with ALPS

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

Strong: Impossible [Brewer00, GilbertLynch02]

Sequential : Impossible [LiptonSandberg88, AttiyaWelch94]

Causal+ : Causal + Convergent Conflict Handling

COPS

Eventual : Dynamo, Cassandra

Rules for Potential Causality

1. **Execution Thread.** If a and b are two operations in a *single thread of execution*, then $a \rightsquigarrow b$ 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 $a \rightsquigarrow b$.
3. **Transitivity.** For operations a, b, and c, if $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$.

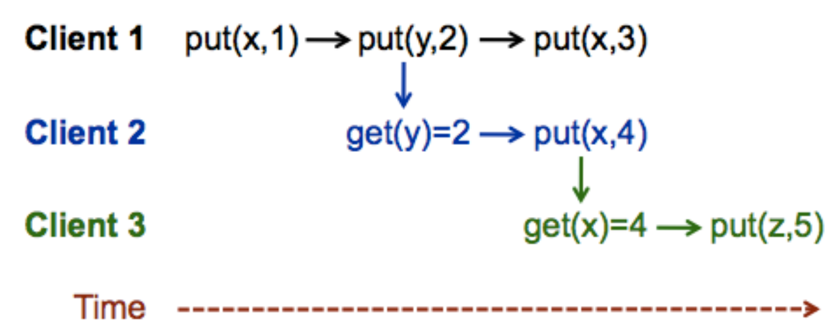
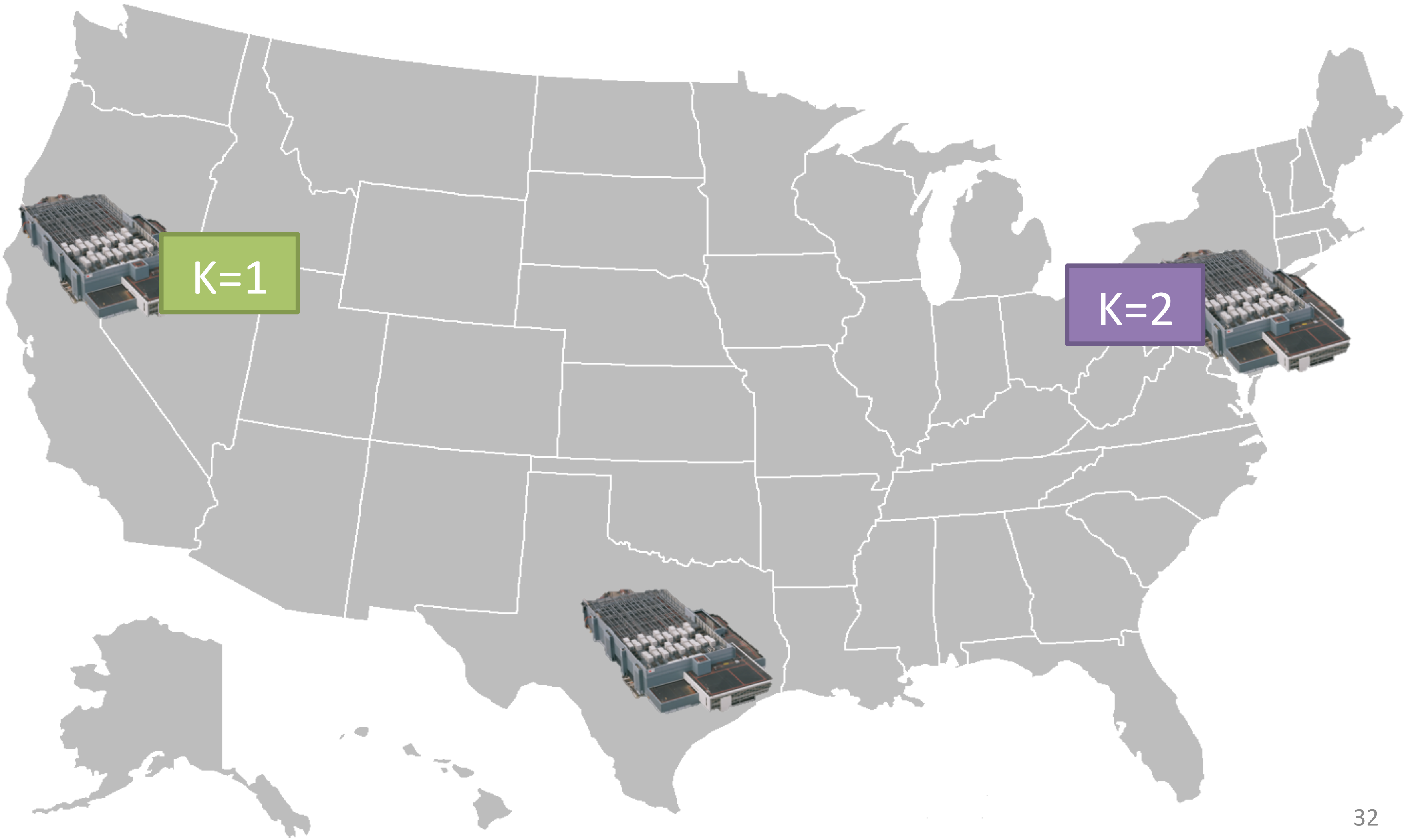


Figure 2: Graph showing the causal relationship between operations at a replica. An edge from a to b indicates that $a \rightsquigarrow b$, or b depends on a.

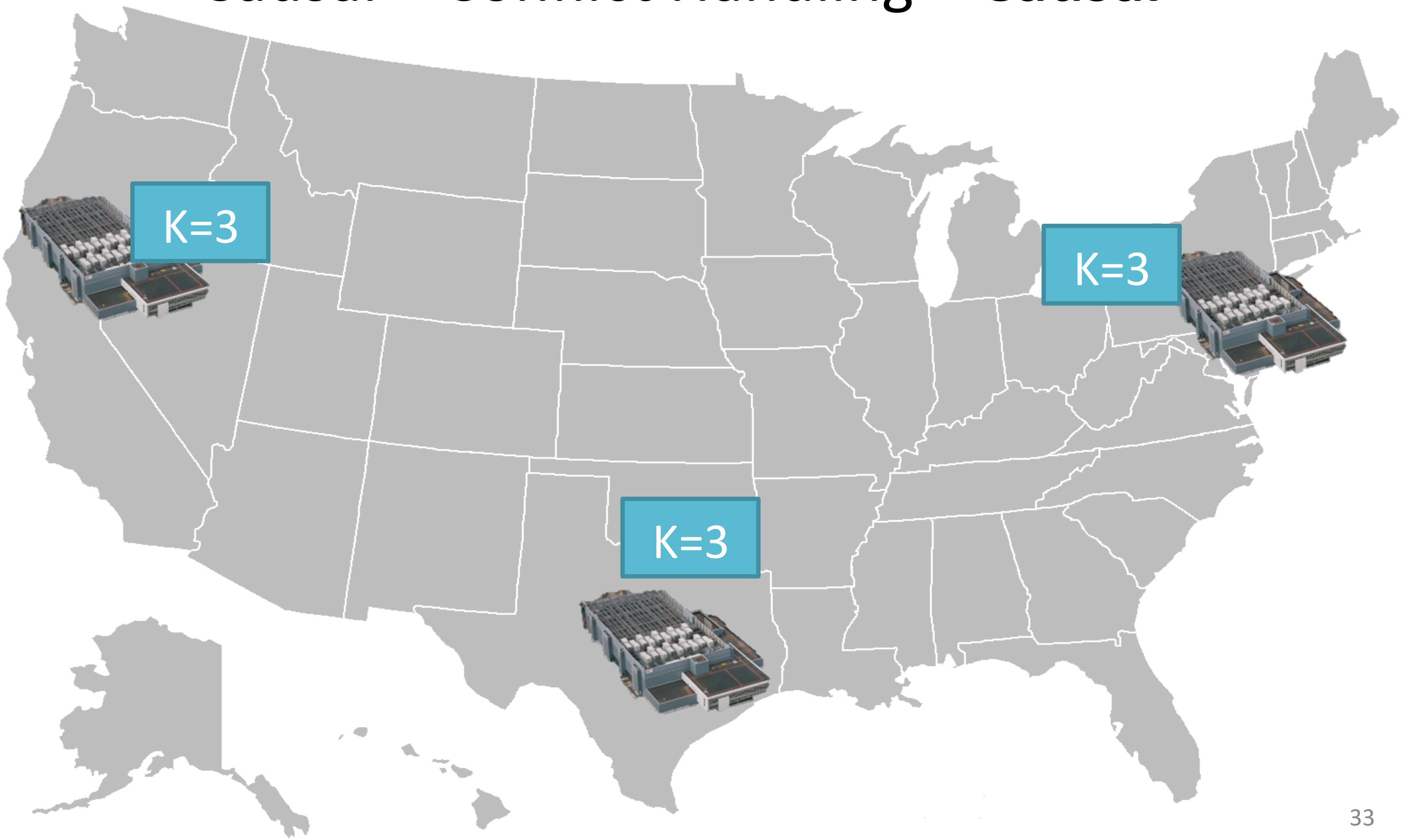
- The value returned by a get operation has to be consistent with the order defined by these rules.
- It must appear that the operation that writes a value occurs after all operations that causally precede it

Conflicts in Causal



Conflicts in Causal

Causal + Conflict Handling = **Causal+**



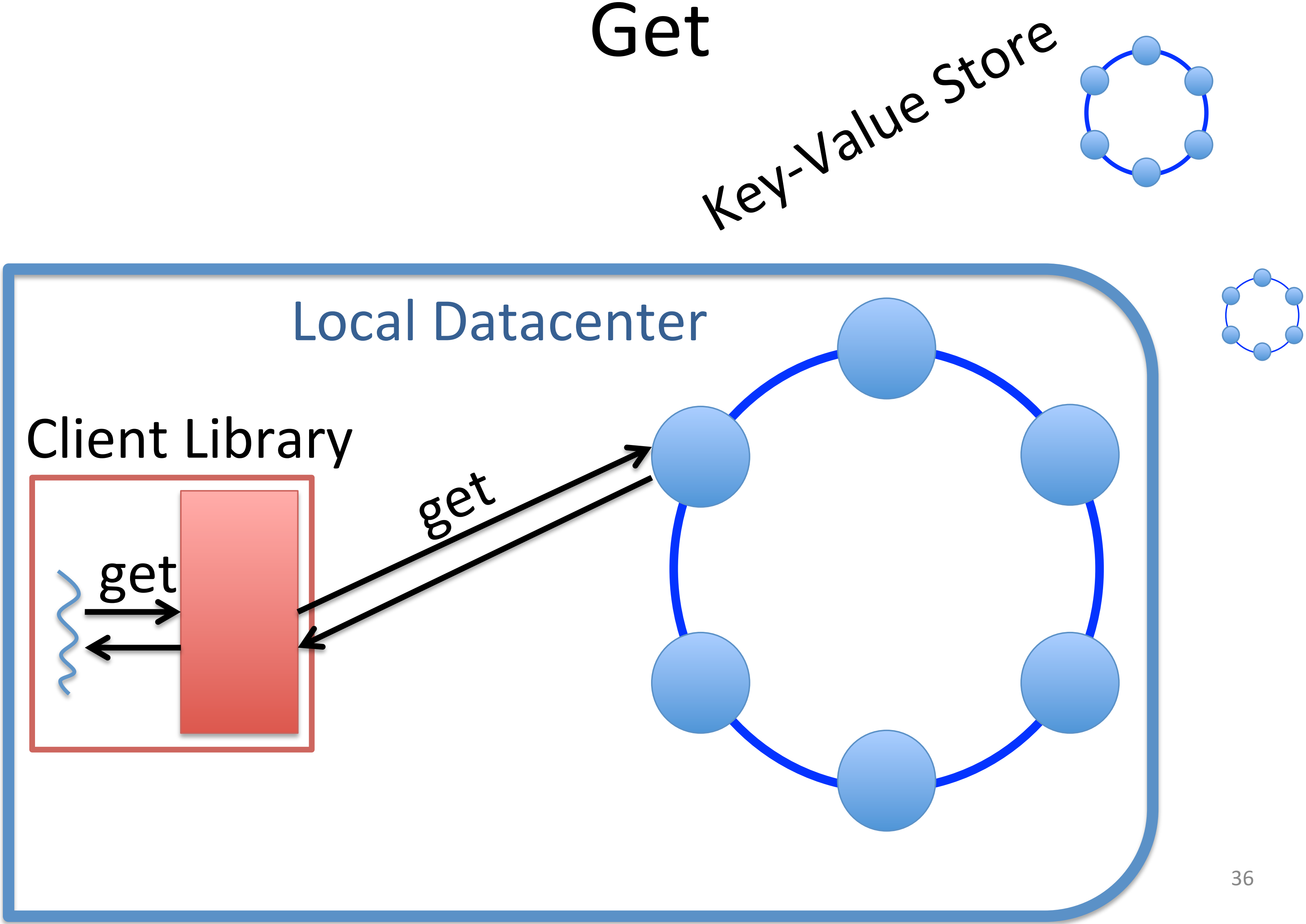
Previous Causal+ Systems

- Bayou ‘94, TACT ‘00, PRACTI ‘06
  - Limited Scalability
  - All data should fit on same machine (Bayou)
  - Data that could be accessed together should be on same machine (PRACTI)
  - Log-exchange based

COPS

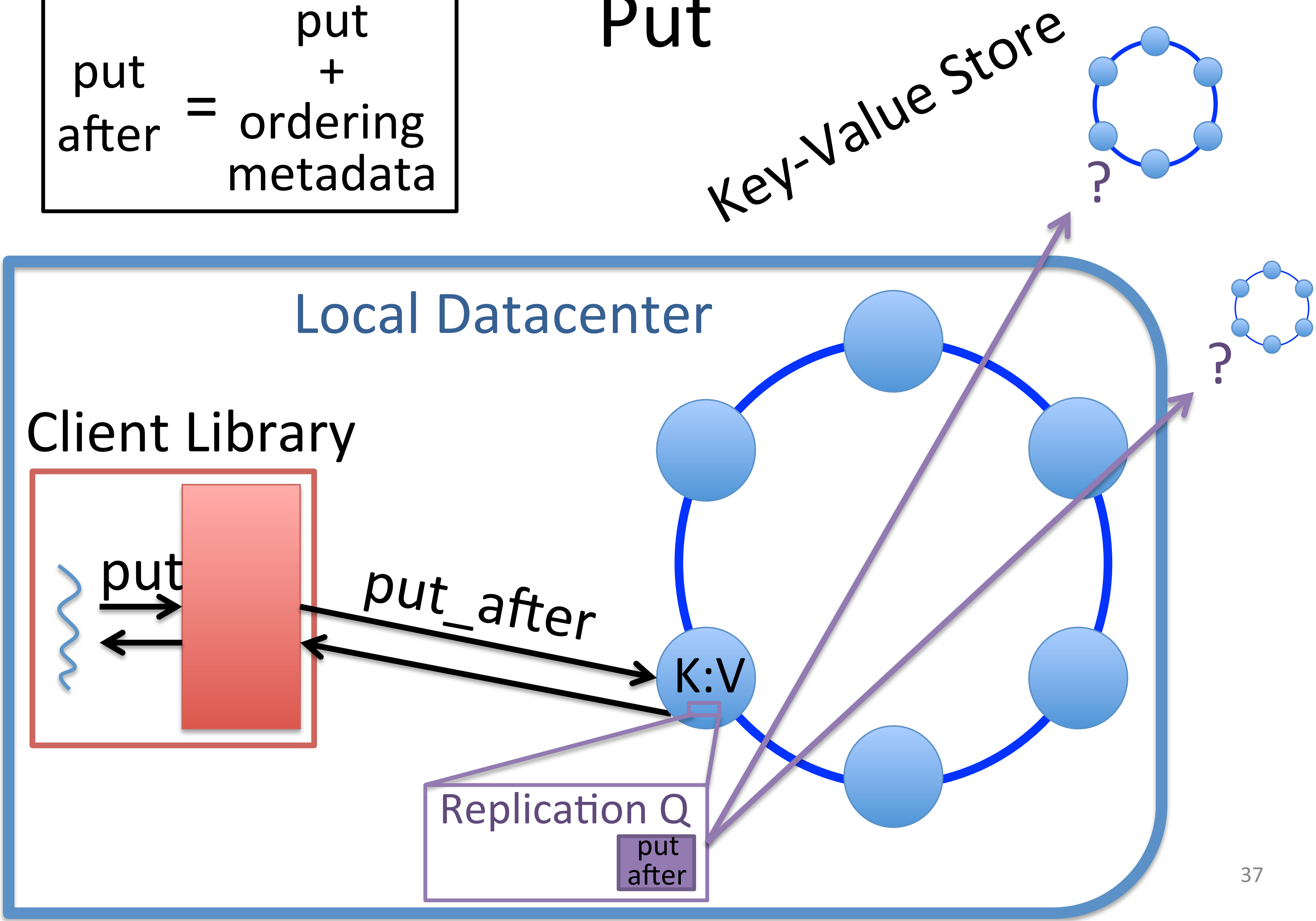
- Dependency metadata explicitly captures causality
- Versions
  - Different values of a key
  - Each replica returns non decreasing versions of a key
- Dependencies
  - y_j depends on x_i if and only if put(x_i) -> put (y_j)
  - Provide causal+ consistency by writing a version only after writing all of its dependencies

Get



Put

put after = put + ordering metadata

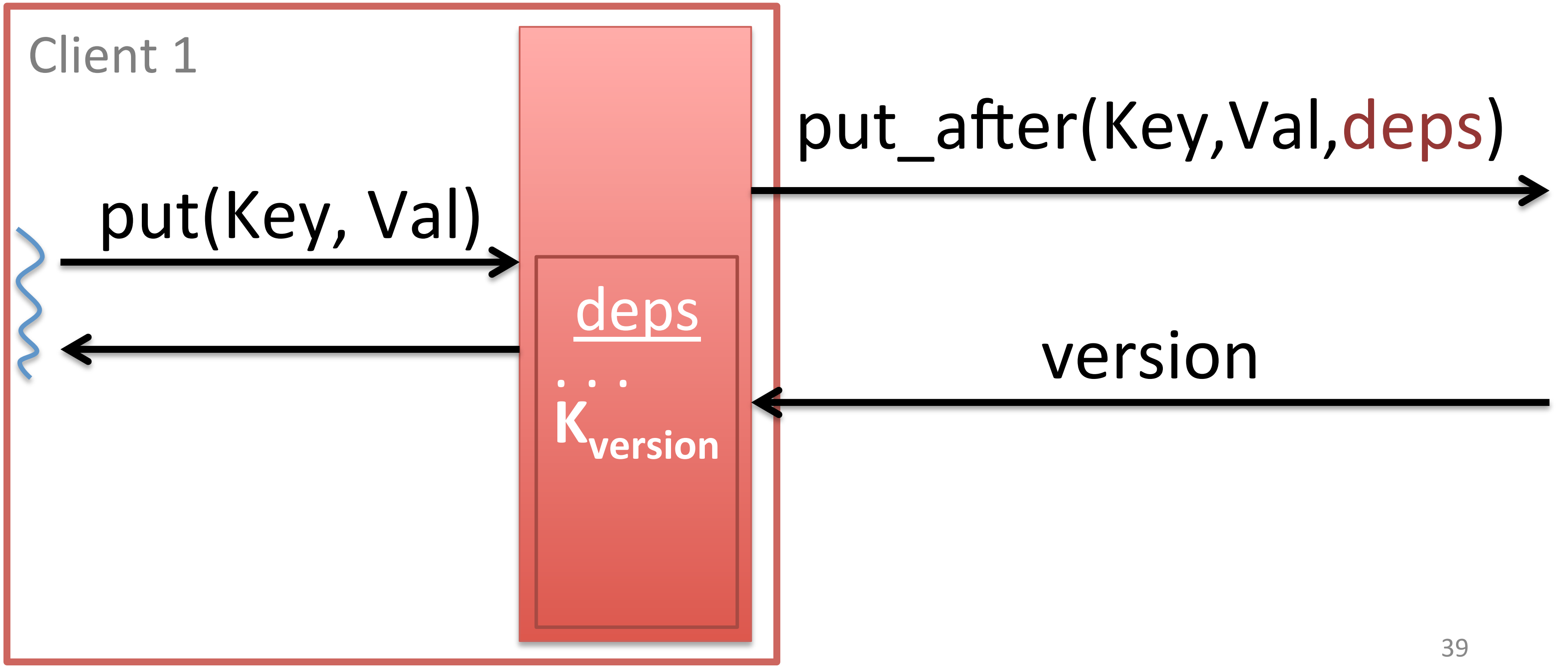


Dependencies

- Dependencies are explicit metadata on values
- Library tracks and attaches them to put_afters

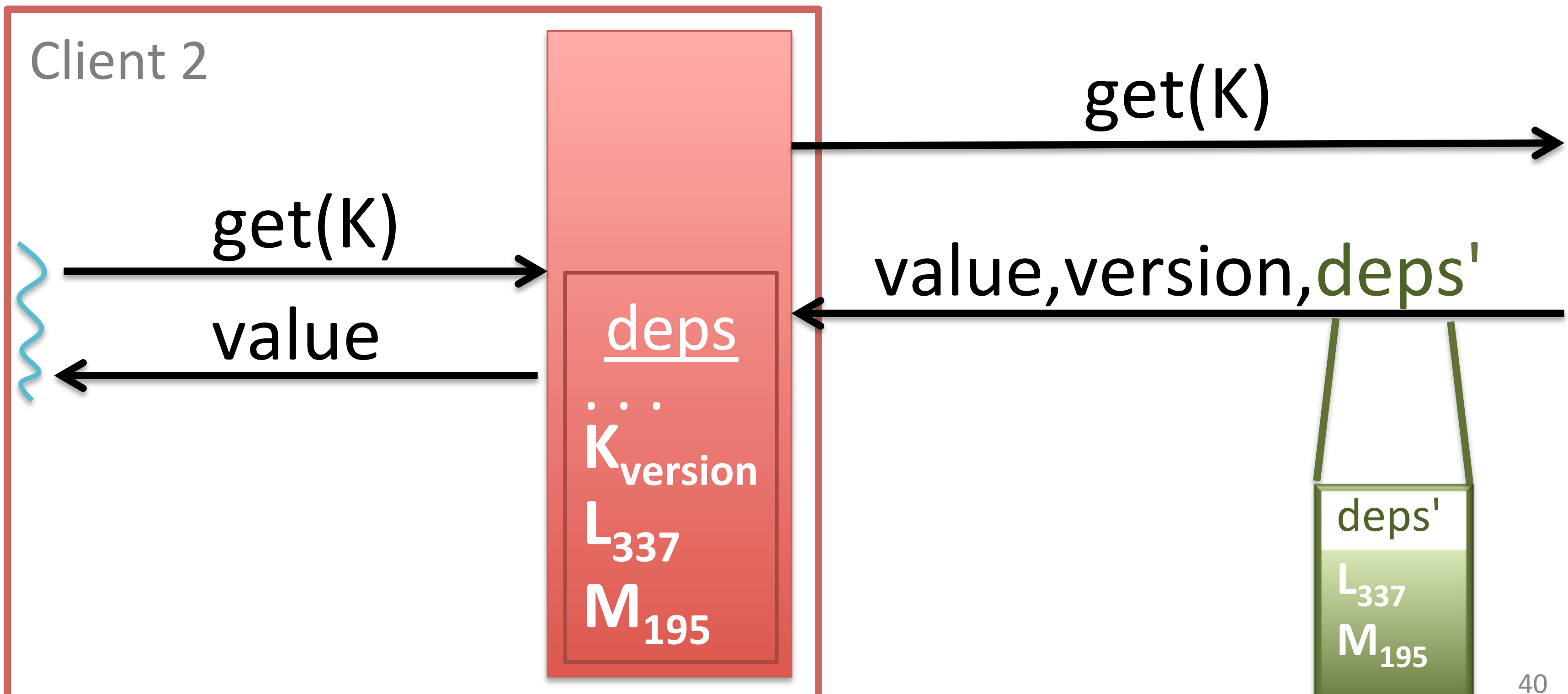
Dependencies

- Dependencies are explicit metadata on values
- Library tracks and attaches them to put_afters

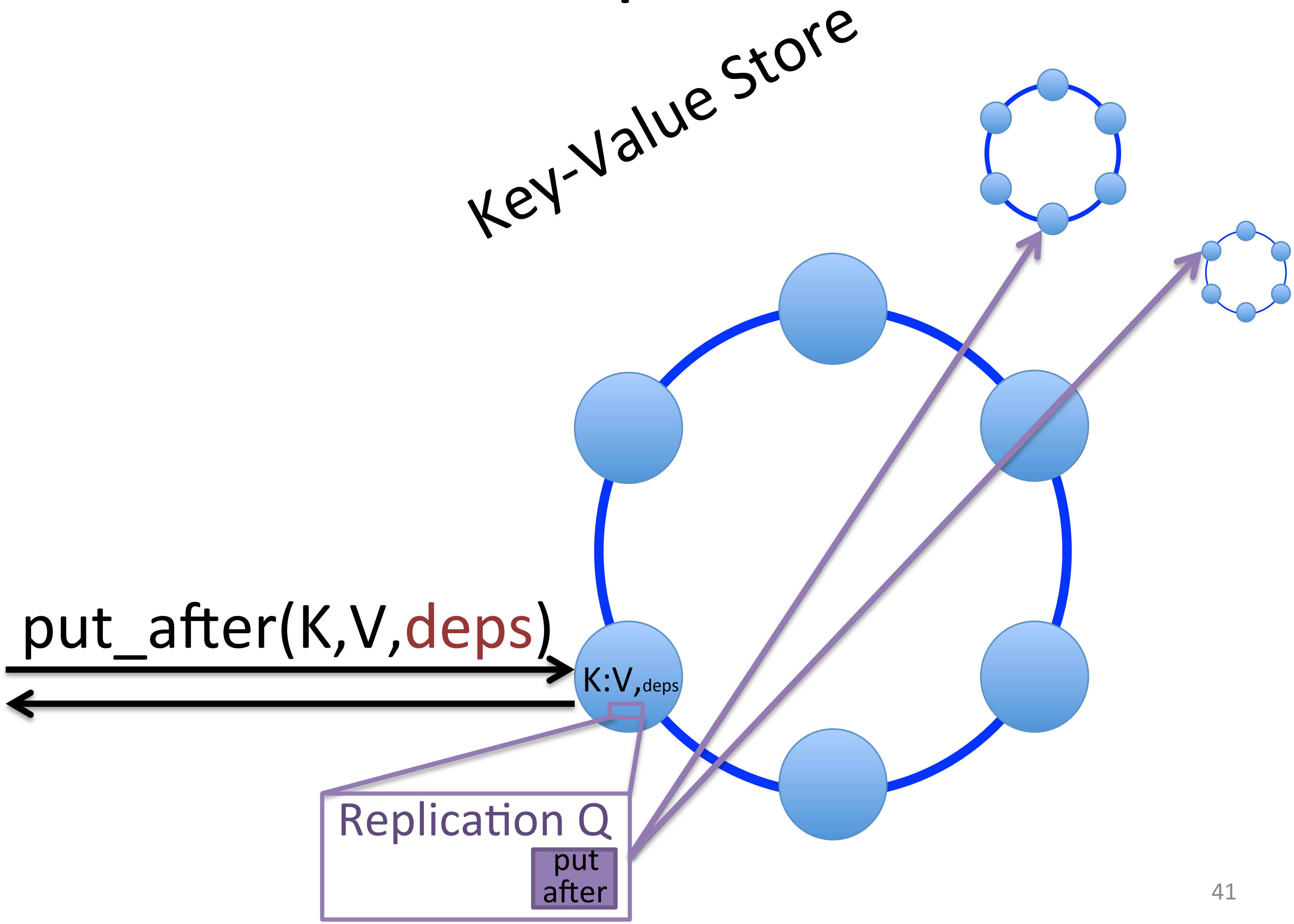


Dependencies

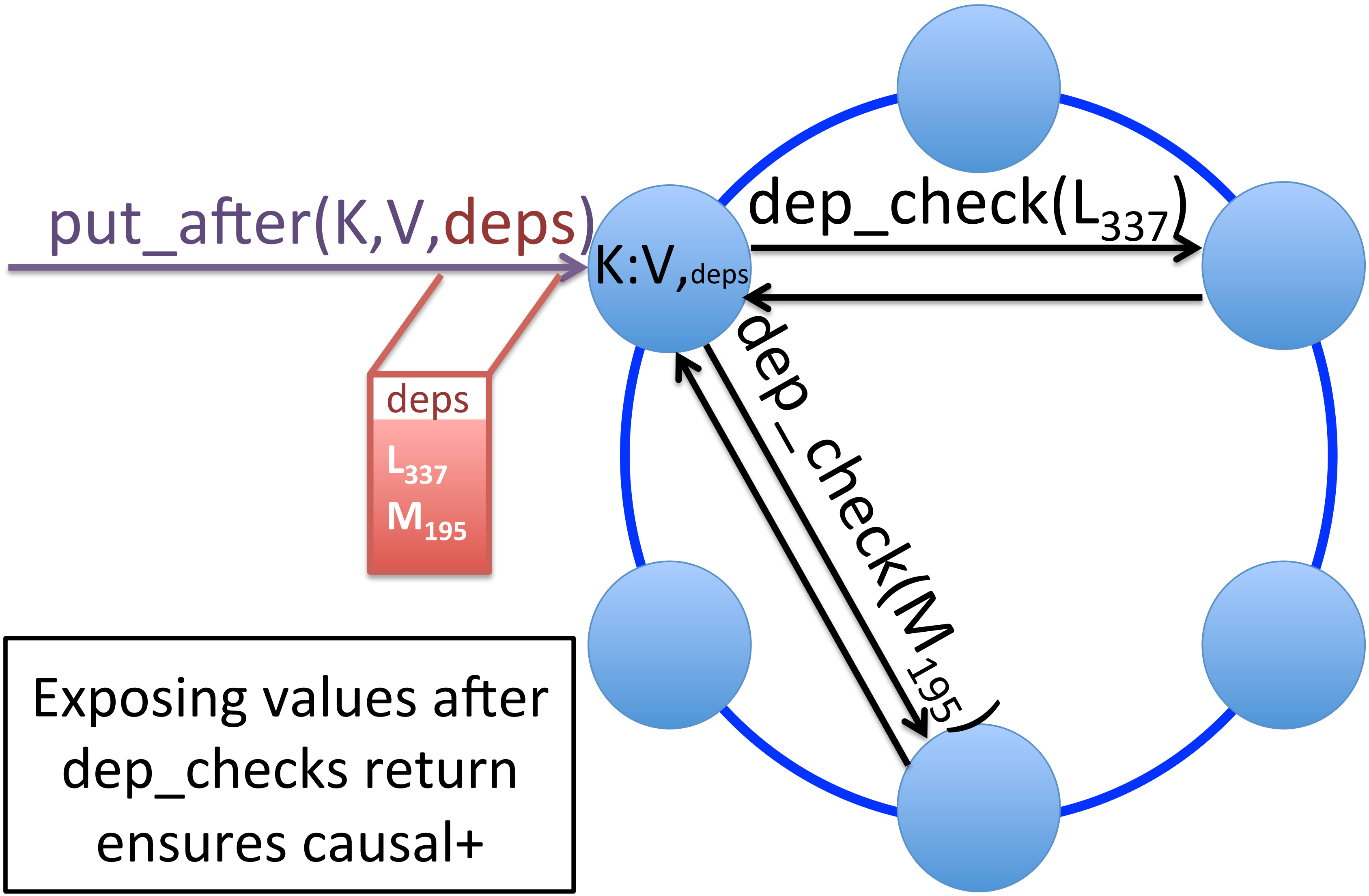
- Dependencies are explicit metadata on values
- Library tracks and attaches them to put_afters



Causal+ Replication



Causal+ Replication

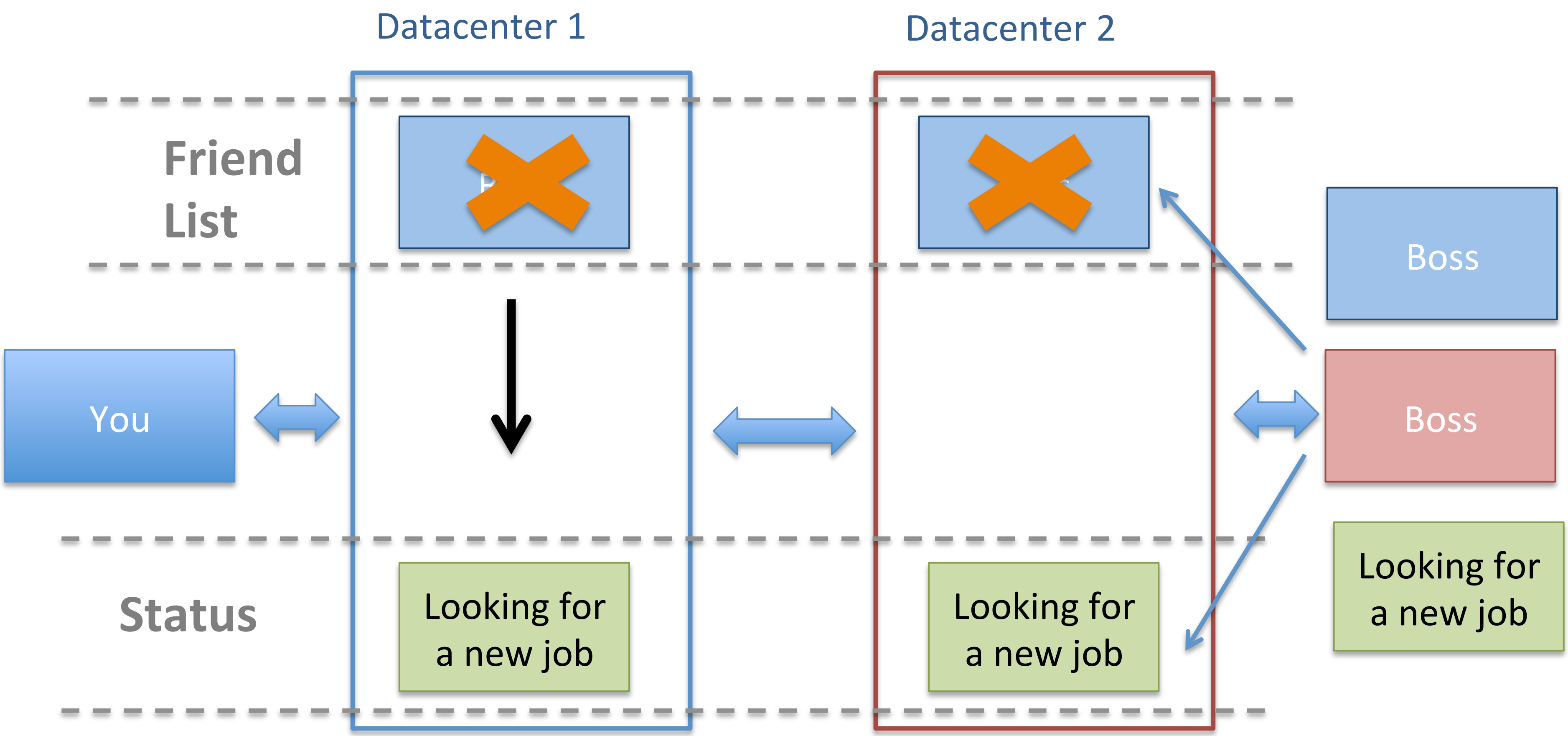


Exposing values after dep_checks return ensures causal+

Basic COPS Summary

- Serve operations locally, replicate in background
  - “Always On”
- Partition keyspace onto many nodes
  - Scalability
- Control replication with dependencies
  - Causal+ Consistency

This Isn’t Enough



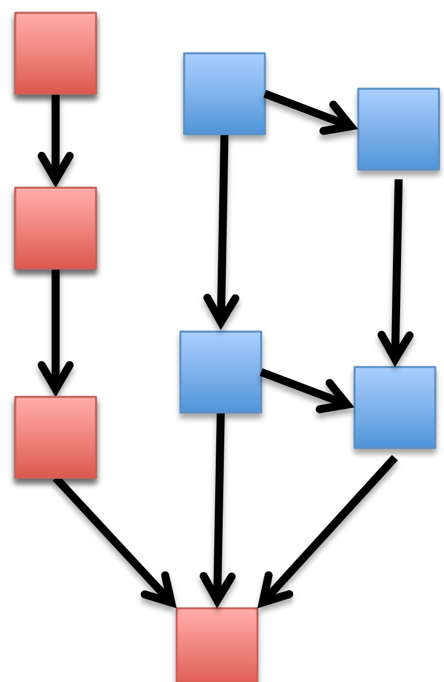
- Get Transactions: Provide consistent view of multiple keys
  - get_trans(key1, key2, key3)

System So Far

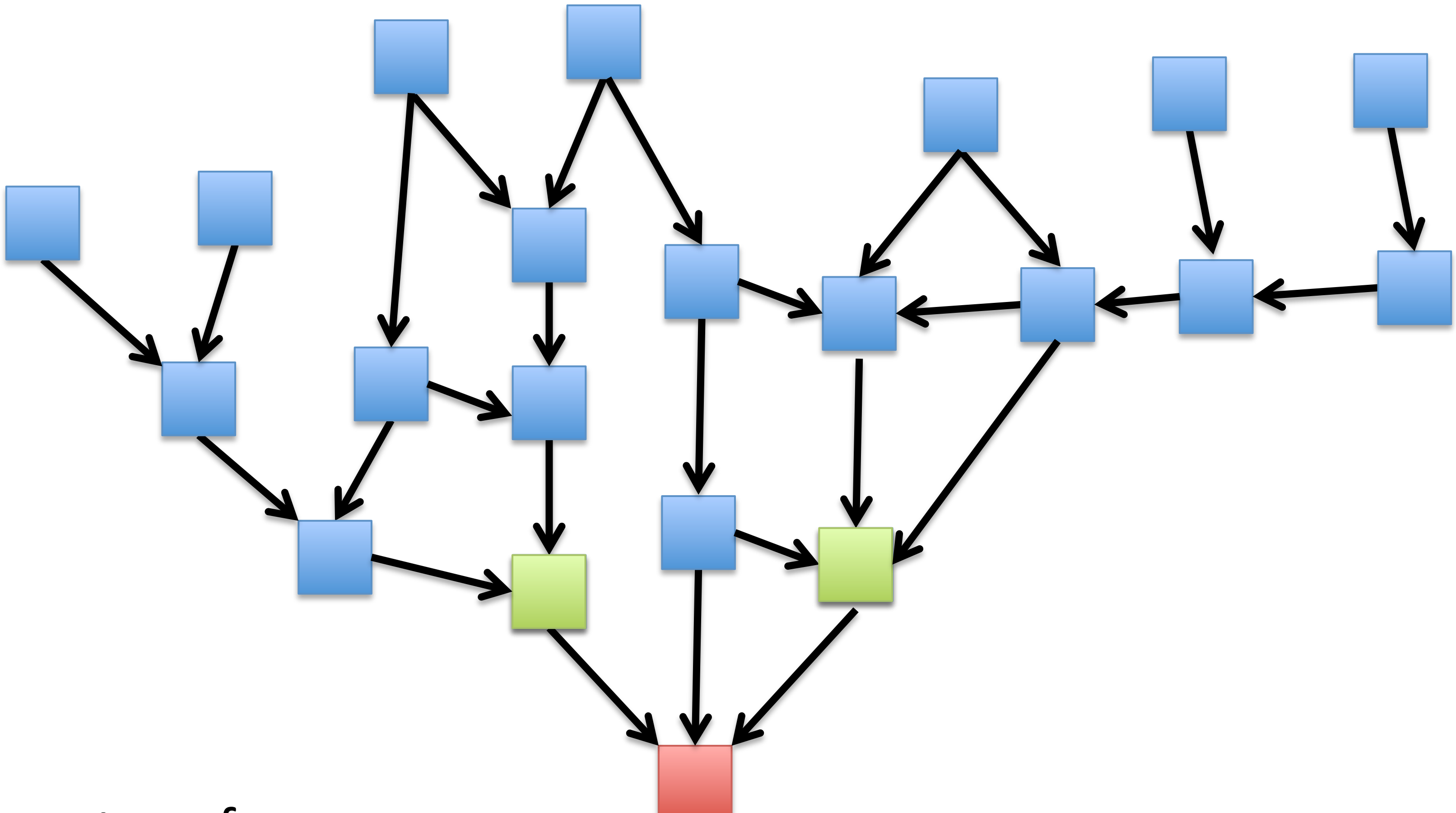
- ALPS and Causal+, but …
- Proliferation of dependencies reduces efficiency
  - Results in lots of metadata
  - Requires lots of verification

Many Dependencies

- Dependencies grow with client lifetime



Nearest Dependencies



- Nearest are few
- Only check nearest when replicating

Other Mechanisms

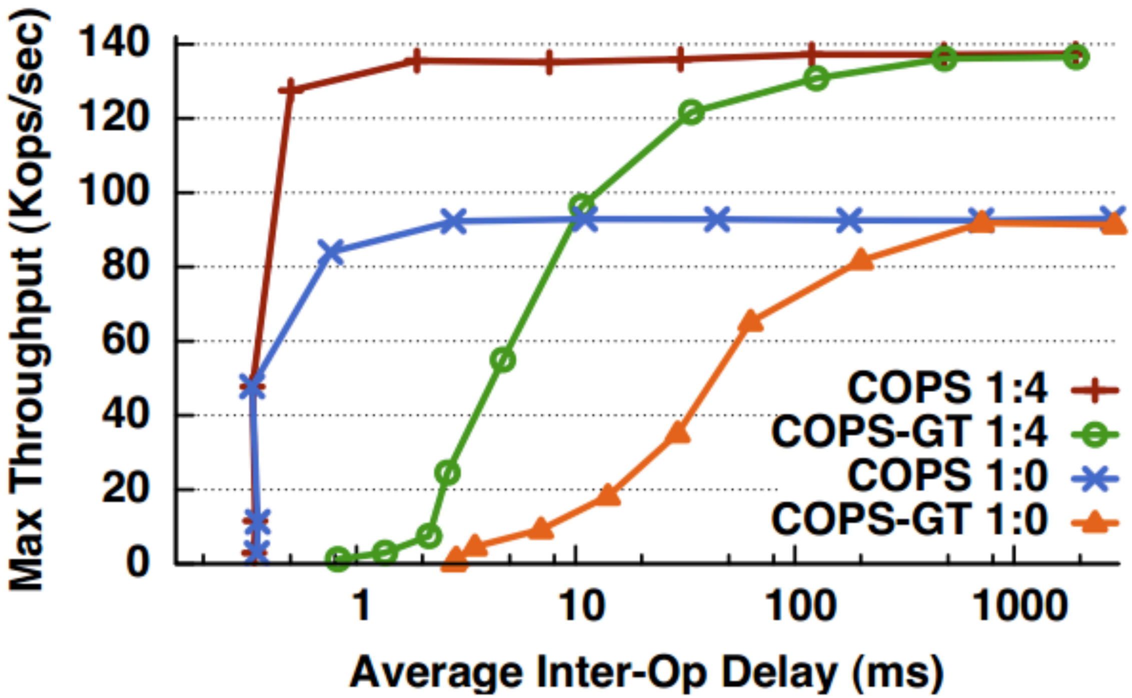
- Garbage Collection Subsystem
  - Reduce the amount of extra state in the system
- Fault Tolerance
  - Clients, nodes and datacenter failures
- Conflict Detection Mechanisms

Latency and Throughput

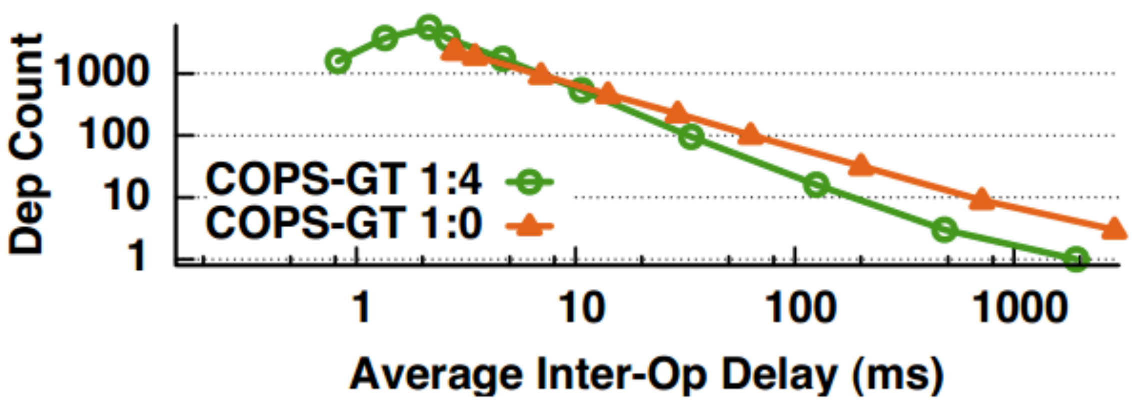
System	Operation	Latency (ms)			Throughput
		50%	99%	99.9%	(Kops/s)
Thrift	ping	0.26	3.62	12.25	60
COPS	get_by_version	0.37	3.08	11.29	52
COPS-GT	get_by_version	0.38	3.14	9.52	52
COPS	put_after (1)	0.57	6.91	11.37	30
COPS-GT	put_after (1)	0.91	5.37	7.37	24
COPS-GT	put_after (130)	1.03	7.45	11.54	20

Table 2: Latency (in ms) and throughput (in Kops/s) of various operations for 1B objects in saturated systems. put_after(x) includes metadata for x dependencies.

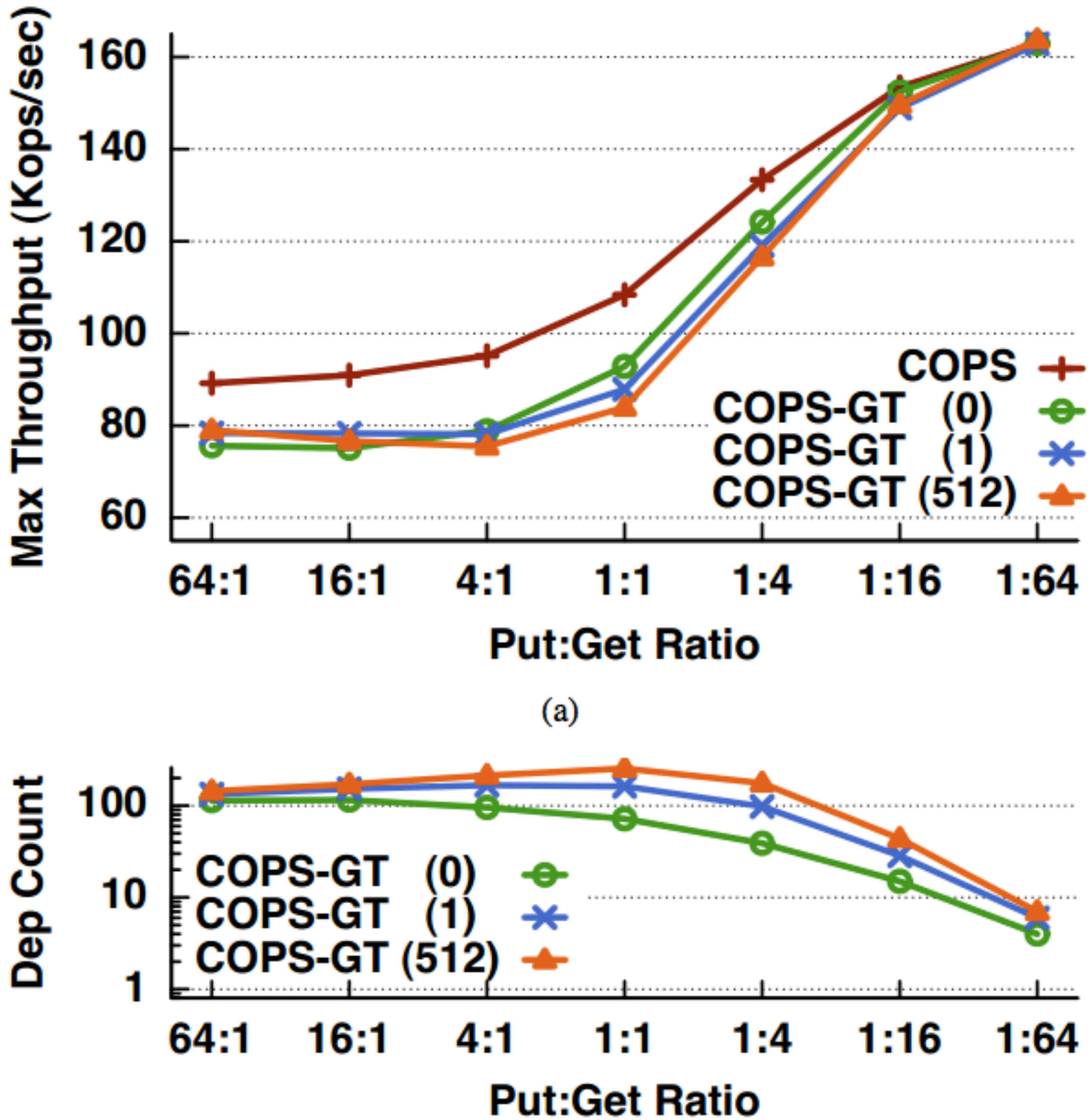
Sensitivity Analysis



(a)



Sensitivity Analysis



COPS Scales Out

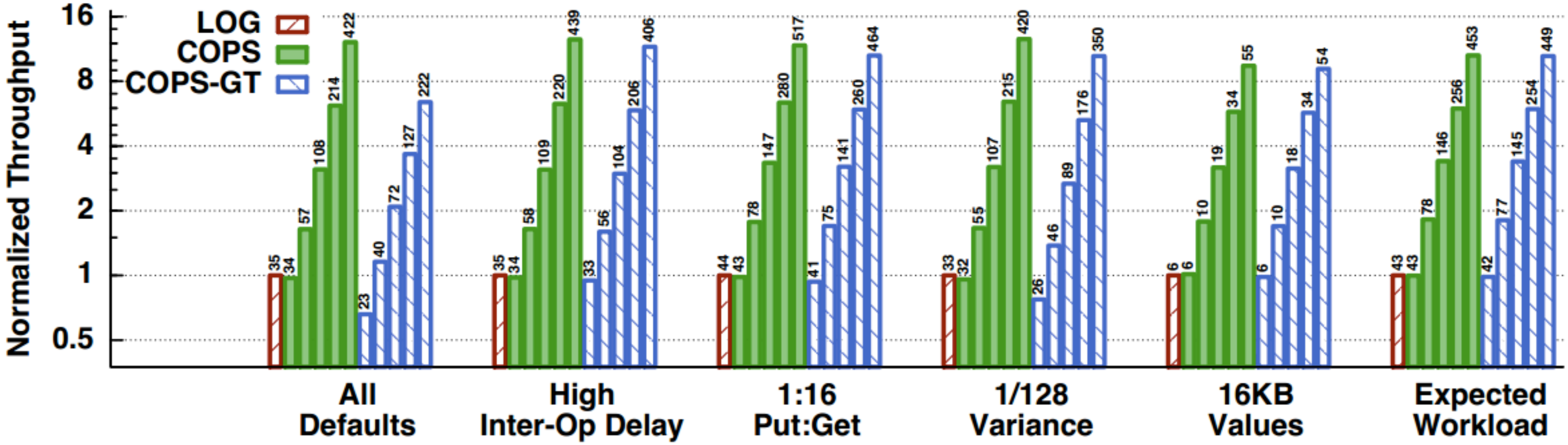


Figure 12: Throughput for LOG with 1 server/datacenter, and COPS and COPS-GT with 1, 2, 4, 8, and 16 servers/datacenter, for a variety of scenarios. Throughput is normalized against LOG for each scenario; raw throughput (in Kops/s) is given above each bar.

Comments

- Suggests keeping metadata to track dependencies. Isn't this what Lamport timestamps wanted to avoid?

- Scalability benefits are not clear
  - No comparison with other systems
  - No WAN delays
  - No comparison with eventually consistent data stores

- Will it require each new application that adopts COPS to create a new client library?

Backup Slides

COPS

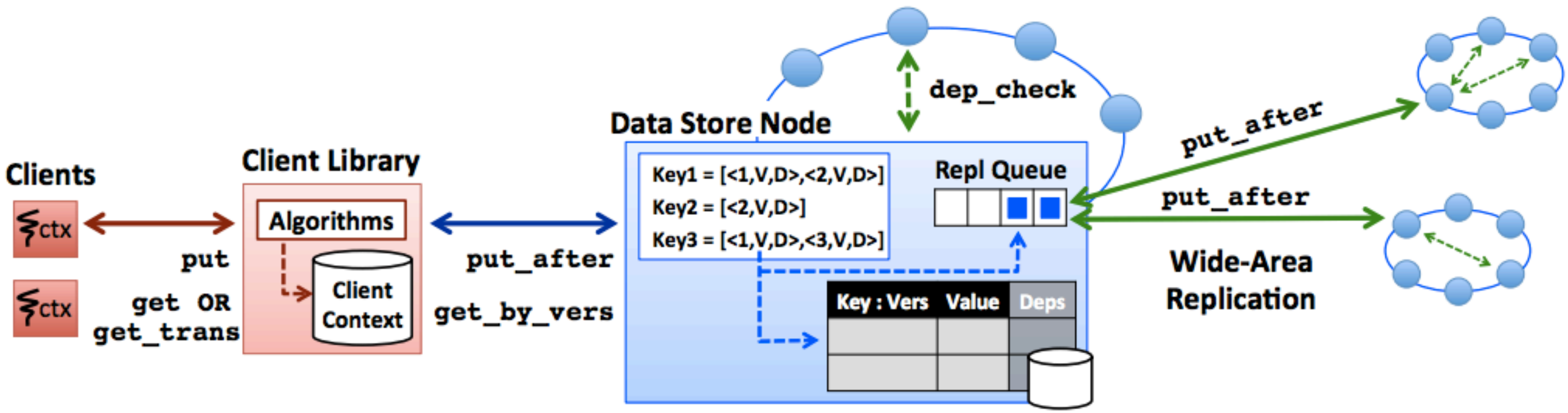
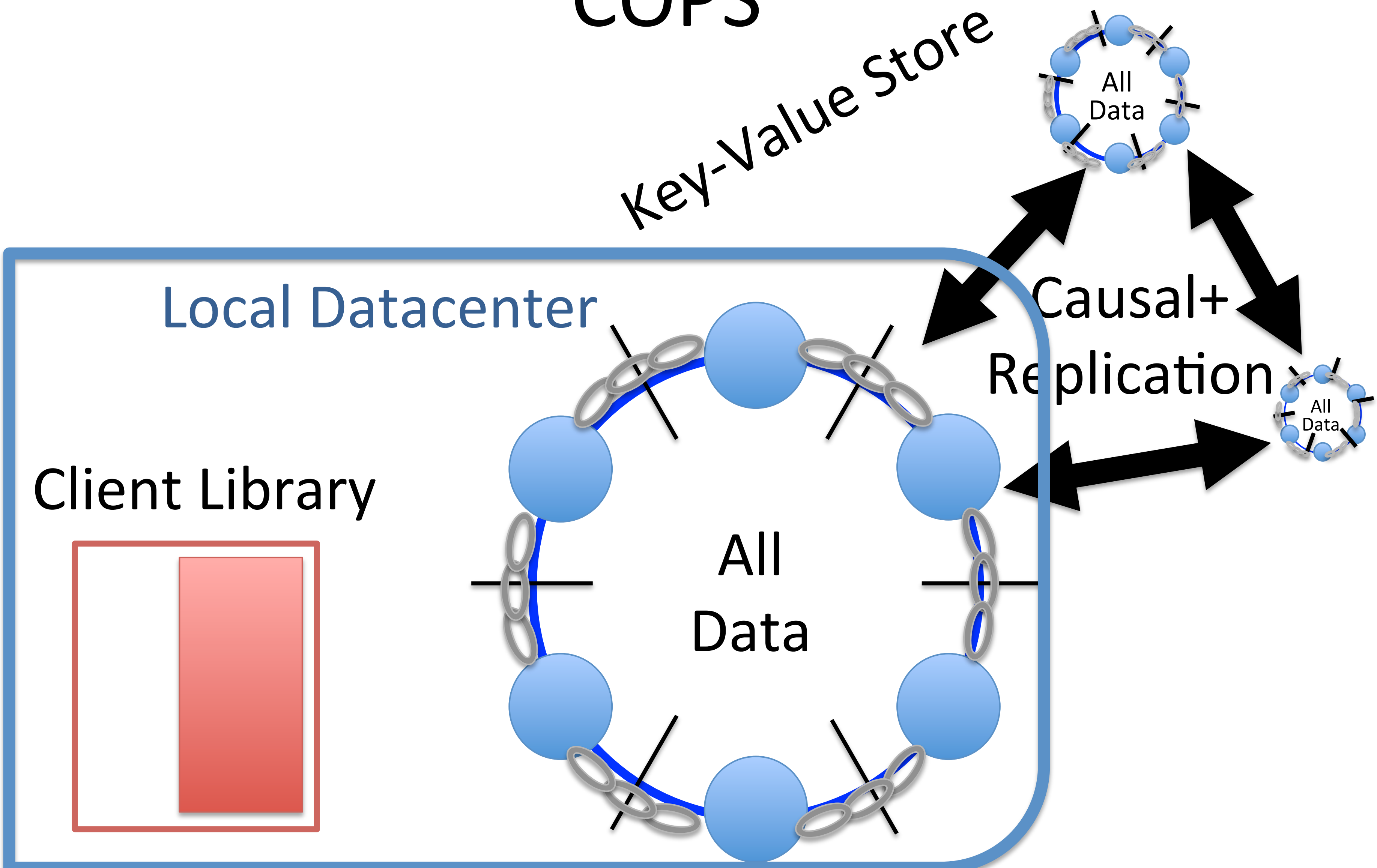


Figure 4: The COPS architecture. A client library exposes a put/get interface to its clients and ensures operations are properly labeled with causal dependencies. A key-value store replicates data between clusters, ensures writes are committed in their local cluster only after their dependencies have been satisfied, and in COPS-GT, stores multiple versions of each key along with dependency metadata.

COPS


Key-Value Store
All
Data
Causal+
Replication
All
Data
Local Datacenter
Client Library
All
Data

Latency and Throughput

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

Table 2: Latency (in ms) and throughput (in Kops/s) of various operations for 1B objects in saturated systems. put_after(x) includes metadata for x dependencies.

Experimental Setup

- Built on top of FAWN-KV: linearizable KV store within a local cluster
- Does not do :
  - Chain Replication
  - Conflict Detection
- Experiments in a single cluster with multiple logical datacenters

System	Causal+	Scalable	Get Trans
LOG	Yes	No	No
COPS	Yes	Yes	No
COPS-GT	Yes	Yes	Yes

Table 1: Summary of three systems under comparison.