Consistency-Based SLA for Cloud Storage

Patryk Hes
Distributed Systems course, 2016

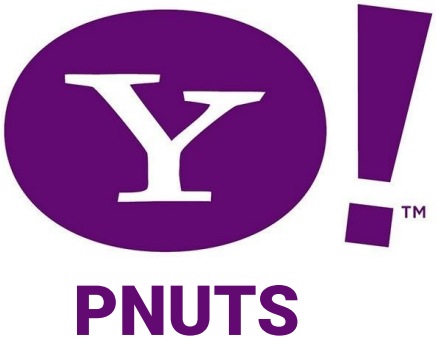
Based on:

Consistency-Based Service Level Agreements for Cloud Storage

Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, Hussam Abu-Libdeh

Cloud Storage

- We consider **NoSQL** data stores (key-value)
- Many different solutions available



Trade-offs

- Read operations: **strong** vs **eventual** consistency
- All mentioned solutions support choosing between those two consistencies...

Trade-offs

- Read operations: **strong** vs **eventual** consistency
- All mentioned solutions support choosing between those two consistencies…
  but do developers know what's best for their apps?

Wasted resources

```
data = WeakRead(key);
display(data);
latestdata = StrongRead(key);
if (data != latestdata) {
    display(latestdata);
}
```

SLI, SLO, SLA

definition by examples

- Service level indicator

  website load time

- Service level objective

  Website load time **is under 300ms in 99.9% cases.**

- Service level agreement

  For every minute that website load time is **not** under 300ms in 99.9% cases, **we pay $100**.

SLA components in cloud storage

A handful of facts (as of 2013) should tell us something about the possible components of SLA:

- DynamoDB charges 2× more for consistent reads
- Amazon loses 1% of sales for each 100ms of latency

SLA components in cloud storage

- Let's make an agreement on **read** operations
- Indicators:
  - Consistency (strong > … > eventual)
  - Latency (lower is better)
- Utility measure: how much user is willing to pay for consistency + latency

Pileus

(/ˈpaɪliː.əs/; Latin for *cap*)

source: *youtube.com*

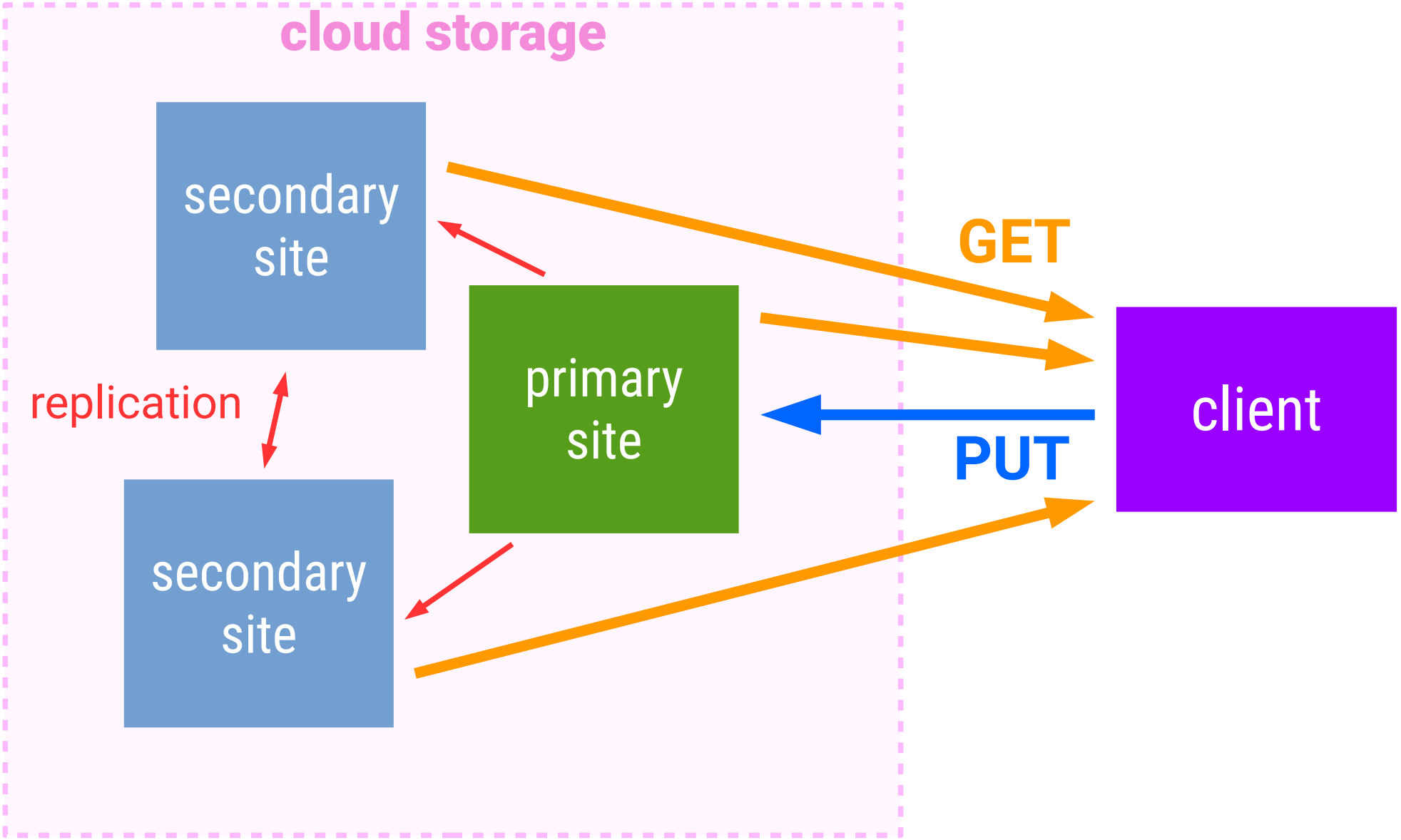
Pileus SLAs

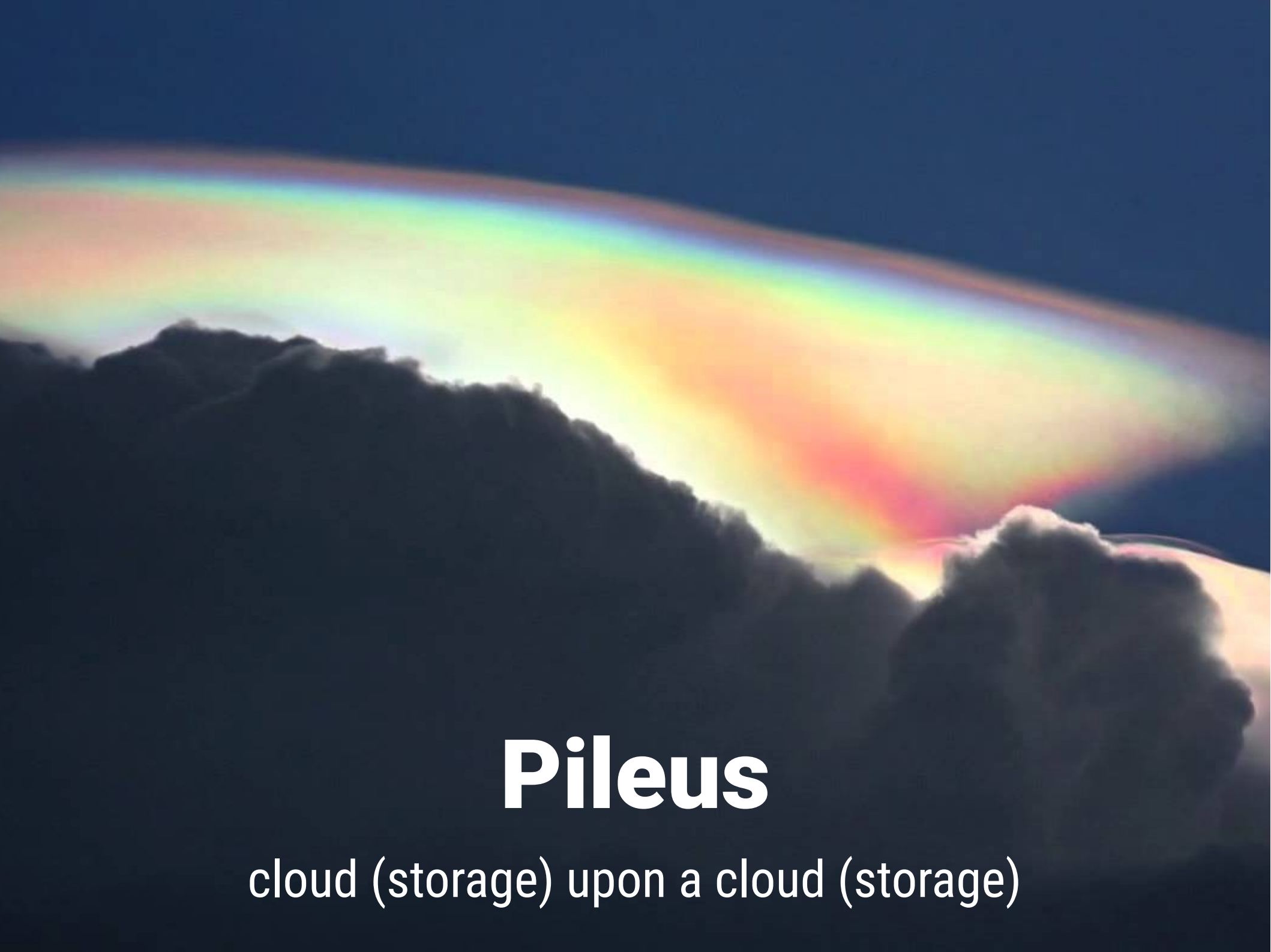
- Don't pick one solution – let SLA be a list of alternative preferences (*sub*-SLAs)
- Preferences should be ordered by utility

Rank	Consistency	Latency	Utility
1.	strong	150ms	1.0
2.	eventual	150ms	0.5
3.	strong	1s	0.25

Pileus architecture

from a bird's eye view





Pileus

cloud (storage) upon a cloud (storage)

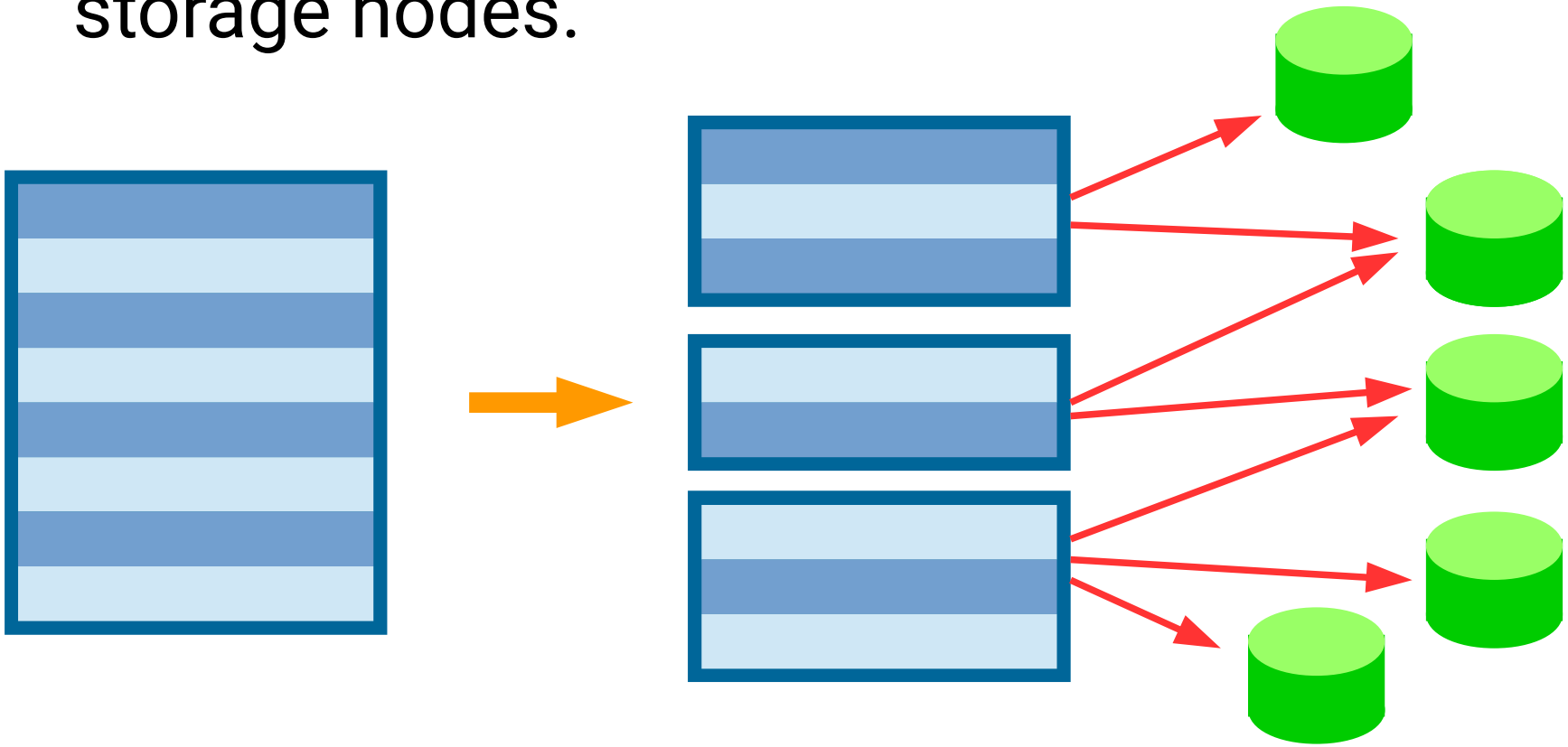
Key-value operations

```
CreateTable(name)
DeleteTable(name)
table = OpenTable(name)
session = BeginSession(table, sla)
EndSession(session)
Put(session, key, value)
value, conditionCode = Get(session, key, sla)
```

Get, **Put** – single-operation transactions

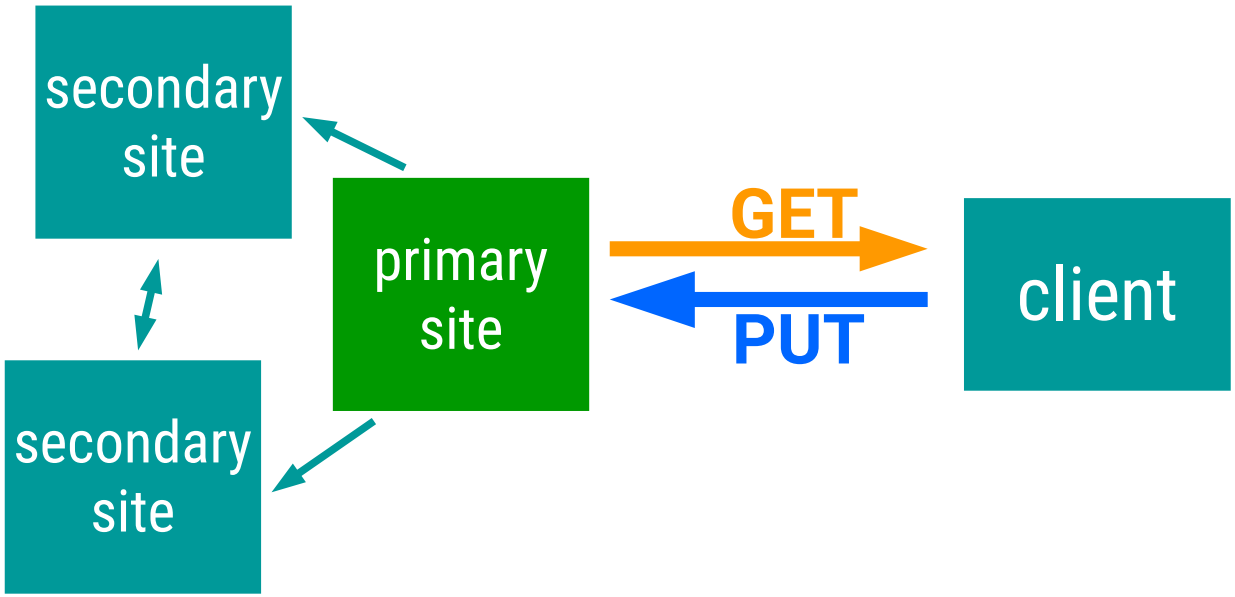
Data sharding

- Big tables can be sharded into smaller *tablets*
- Tablets are the **granularity** of replication and are **independently replicated** on multiple storage nodes.



Data sharding

- Every tablet has its *primary site*
- Site = set of storage nodes within a datacenter

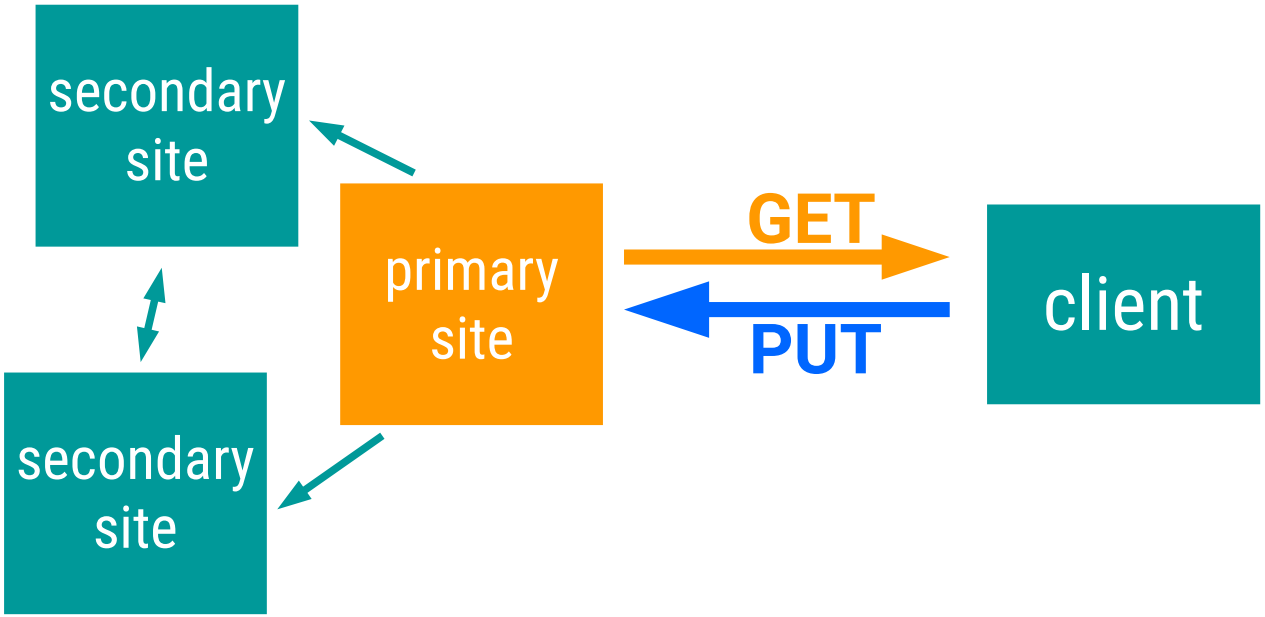


e.g. in Azure: site = three strongly-consistent nodes

Consistency guarantees

strong

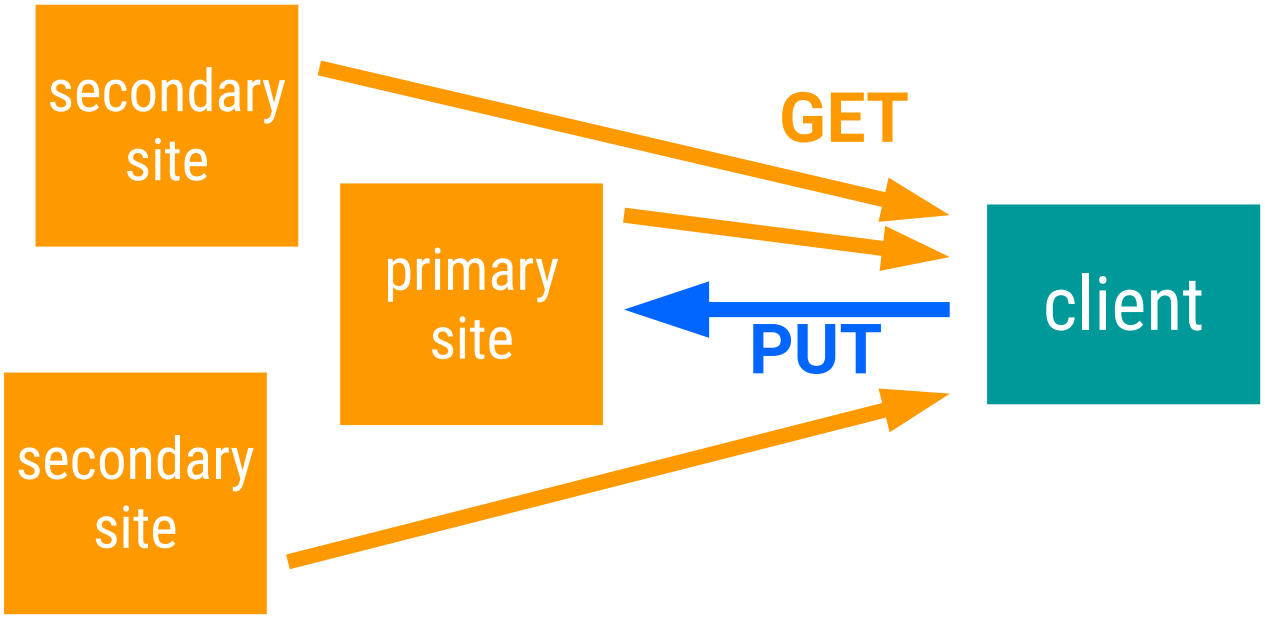
A **Get**(key) returns the value of the last preceding **Put**(key) performed by any client.



Consistency guarantees

eventual

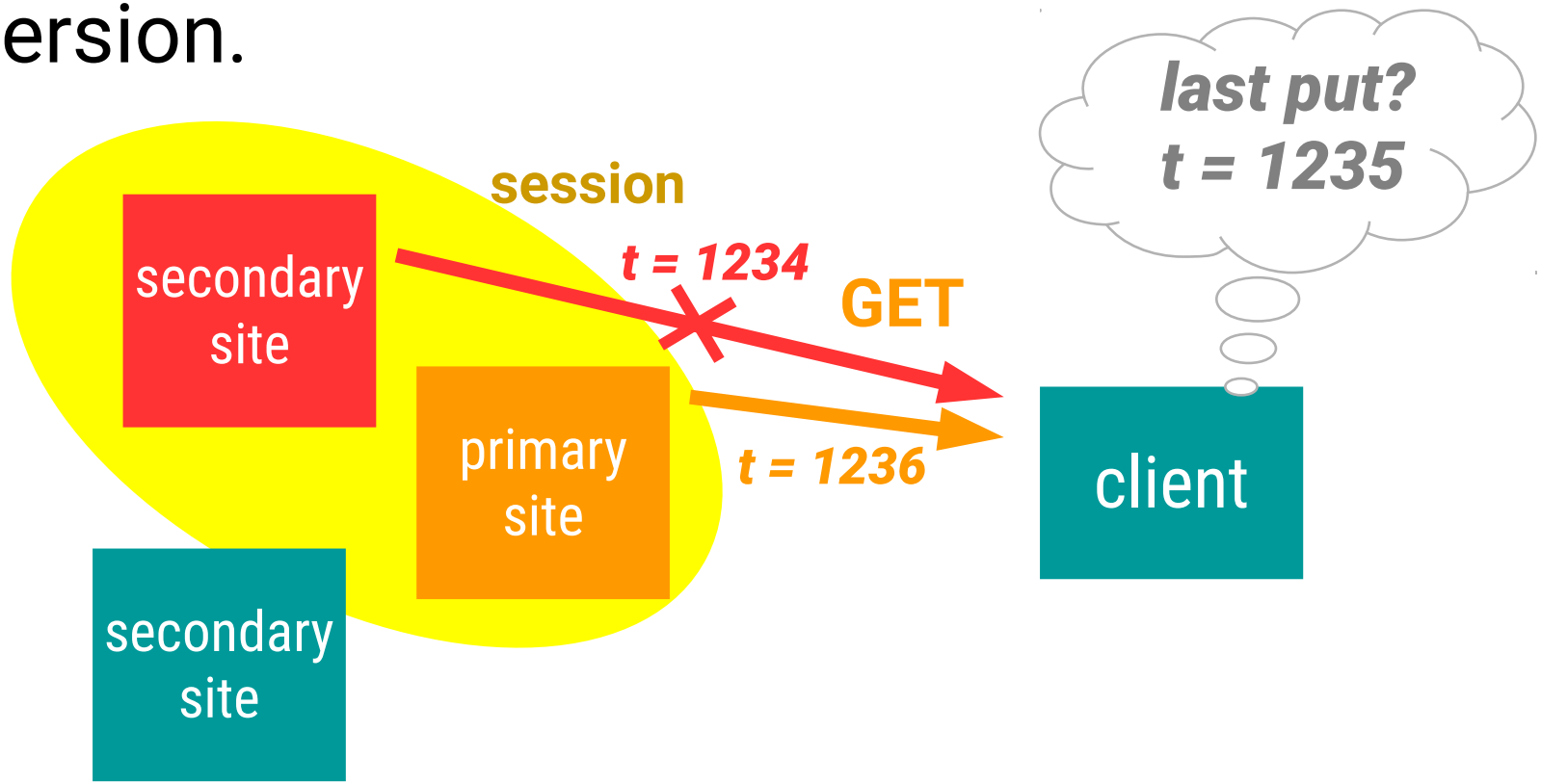
A **Get**(key) returns the value written by any **Put**(key).



Consistency guarantees

read-my-writes

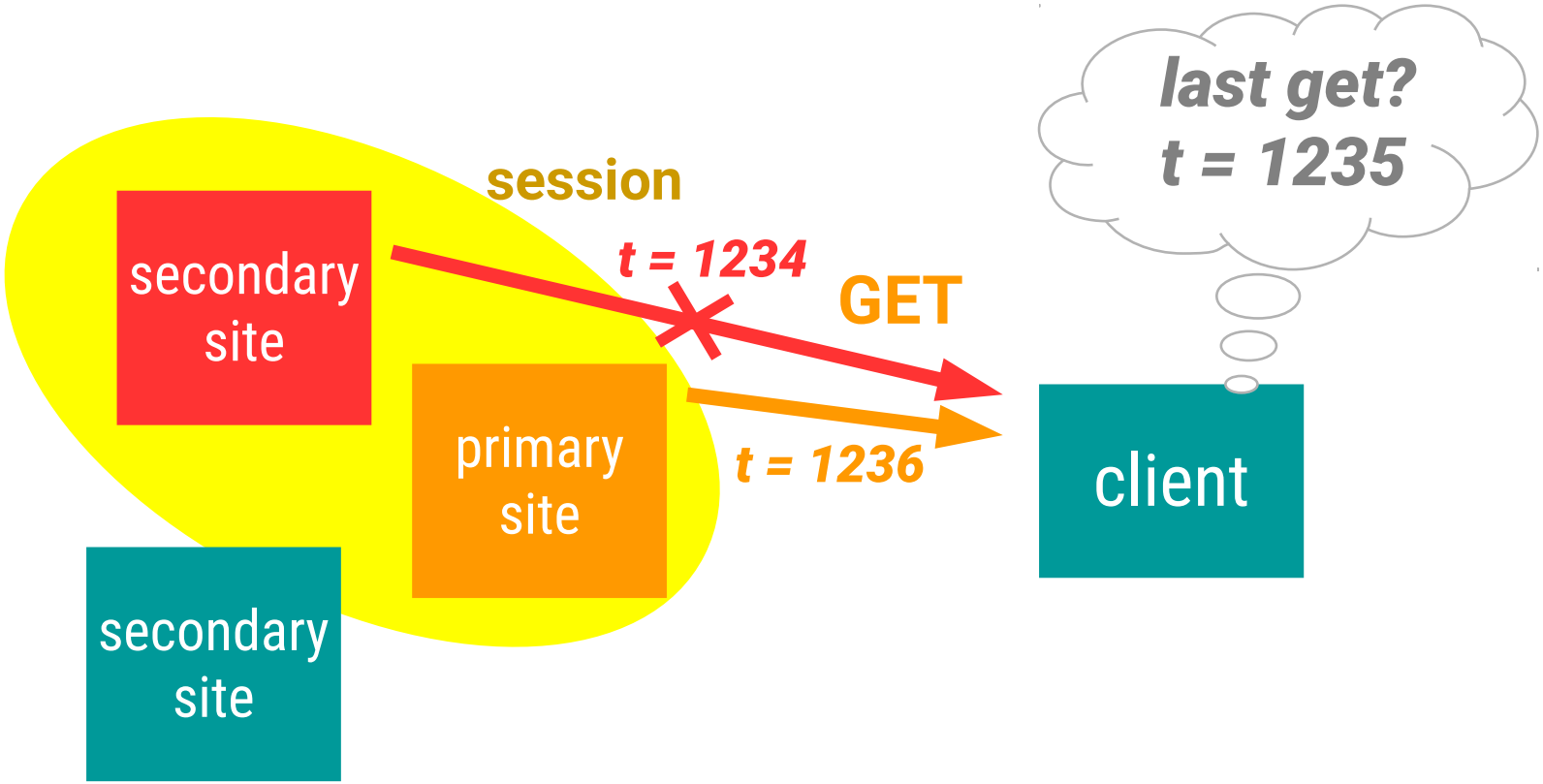
A **Get**(key) returns the value written by the last preceding Put(key) in the same session or returns a later version.



Consistency guarantees

monotonic

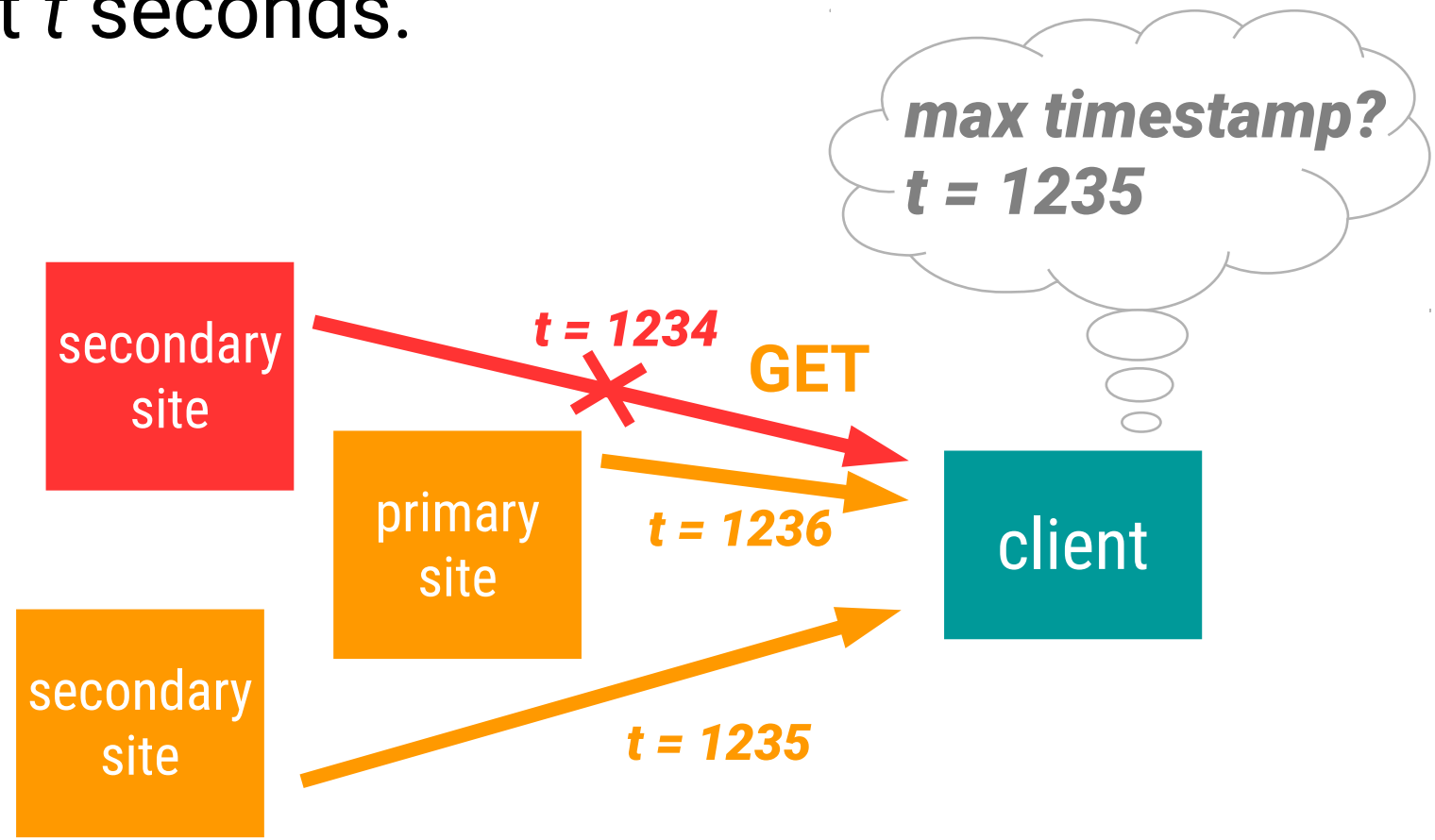
A **Get**(key) returns the same or a later version as a previous **Get**(key) in this session.



Consistency guarantees

bounded(t)

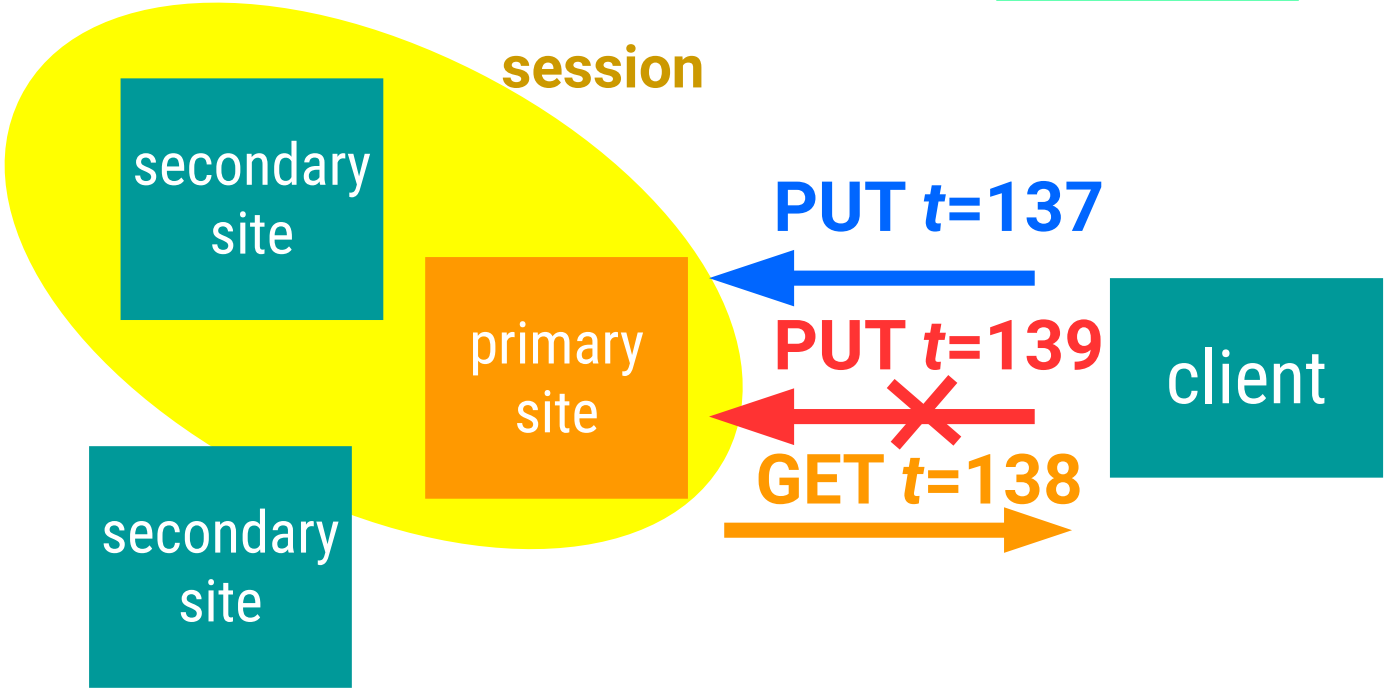
A **Get**(key) returns a value that is stale by at most t seconds.



Consistency guarantees

causal

A **Get**(key) returns the value of a latest **Put**(key) that causally precedes it or returns some later version. The causal precedence relation < is defined such that *op1* < *op2* if **either**:

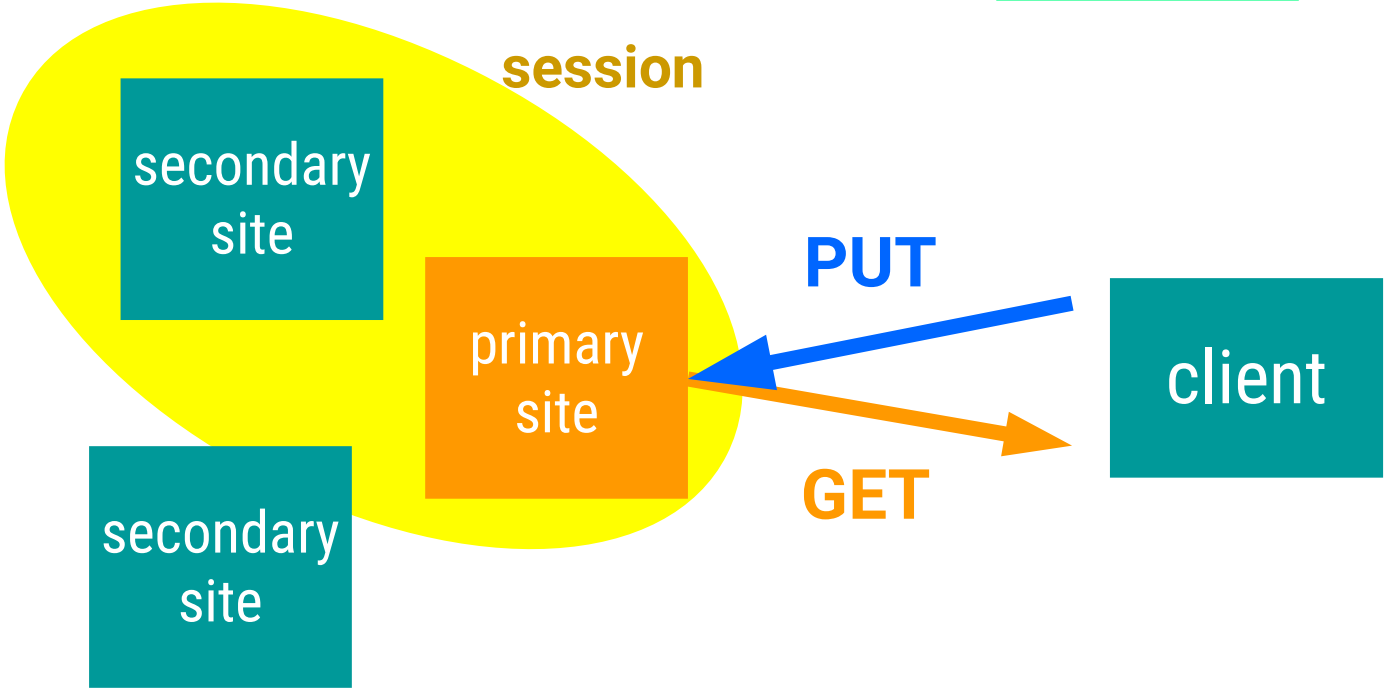


(a) *op1* occurs before *op2* in the same session

Consistency guarantees

causal

A **Get**(key) returns the value of a latest **Put**(key) that causally precedes it or returns some later version. The causal precedence relation < is defined such that *op1* < *op2* if **either**:

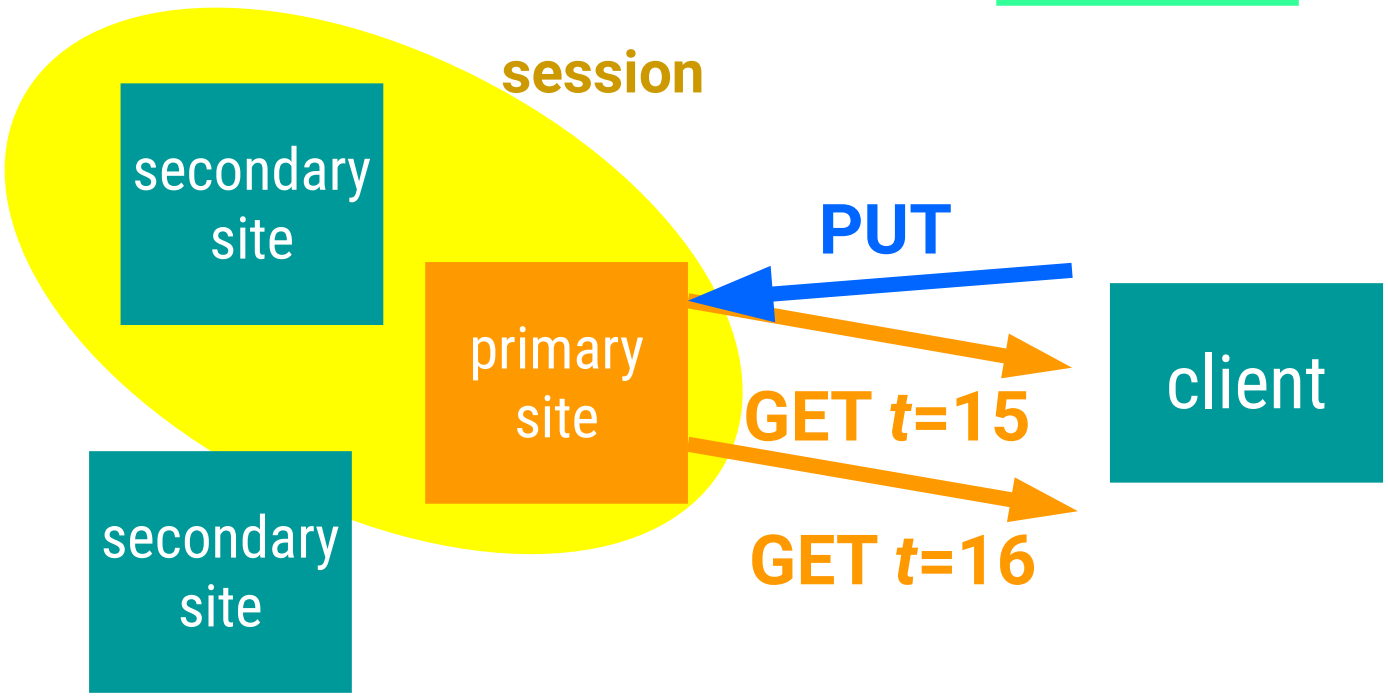


(b) *op1* is a **Put**(key) and *op2* is a **Get**(key) that returns the version put in *op1*

Consistency guarantees

causal

A **Get**(key) returns the value of a latest **Put**(key) that causally precedes it or returns some later version. The causal precedence relation < is defined such that *op1* < *op2* if **either**:



(c) for some *op3*, *op1* < *op3* < *op2*

Wasted resources, revisited

- Simplest way to be sure we get most of SLA: broadcast **Get**s to all sites
- Problem: money
- Pileus approach:

maximize expected utility
while sending a **Get**
to one site at a time

Storage metadata

- State managed by each node:
  - *tablet store* = set of (key, value, timestamp) tuples
  - *high timestamp* = update timestamp of the latest data object version received by node
- All **Put**s assigned increasing timestamp on primary site
- Assuming we can combat network splits and data loss, we have a prefix of the overall **Put** sequence operations on every node.

Client's knowledge

- Client collects the information on highest timestamps of sites when performing **Get**, **Put** – or sometimes, on demand.
- **PNodeCons**(node, consistency, key) – estimate of the probability that the given storage node is sufficiently up-to-date to provide the given consistency guarantee for the given key.
- **PNodeLat**(node, latency) – estimate of the probability that the node can respond to **Get**s within the given response time.
- **PNodeSla** = **PNodeCons** × **PNodeLat**

```
SelectTarget(SLA, key) =
  maxutil = -1;
  bestnodes = {};
  bestlatency = ∞;
  targetSLA = null;

  foreach subSLA in SLA
    foreach node in key.replicas
      util = PNodeSla(node, subSLA.consistency,
                      subSLA.latency, key) * subSLA.utility;
      if (util > maxutil)
        targetSLA = subSLA;
        maxutil = util;
        bestnodes = node;
      else if (util = maxutil)
        bestnodes = bestnodes + node;

  foreach node in bestnodes
    if (node.latency < bestlatency)
      bestnodes = node;
      bestlatency = node.latency;
  return targetSLA, bestnodes;
```

Experimental set-up

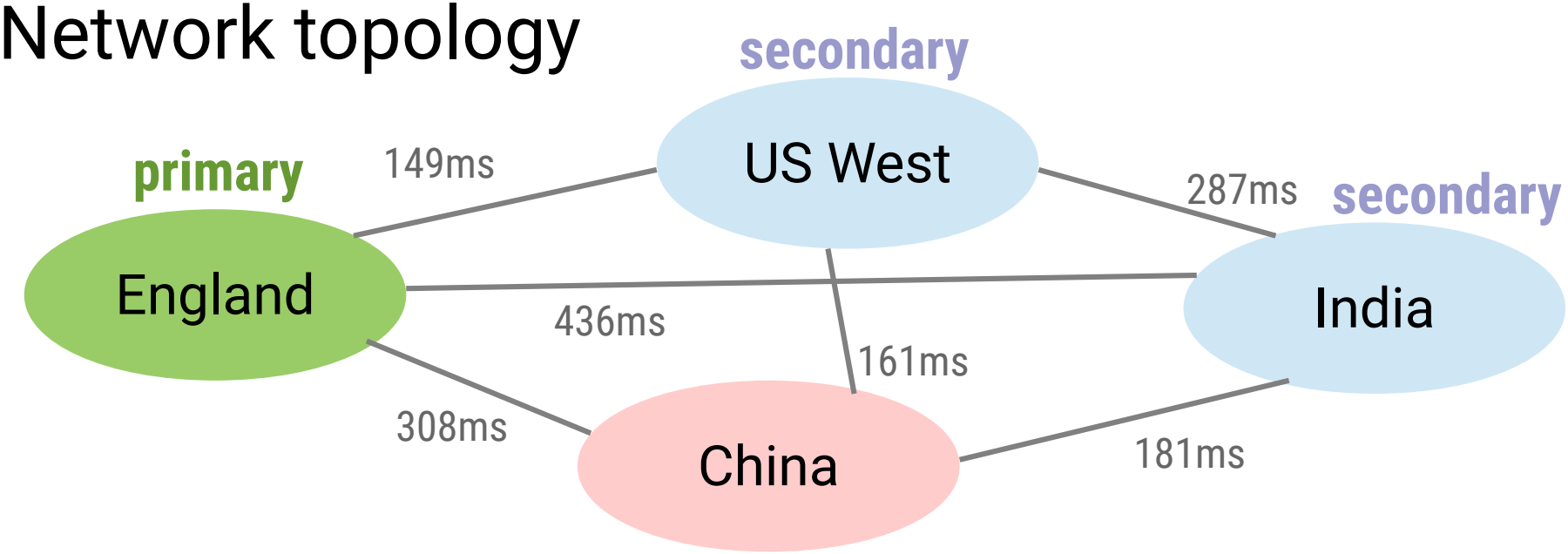
- Workload = Yahoo! Cloud Serving Benchmark
- Clients perform equal numbers of **Put**s and **Get**s to collection of 10k keys
- Session = 400 **Get**s and **Put**s

Experimental set-up: shopping cart

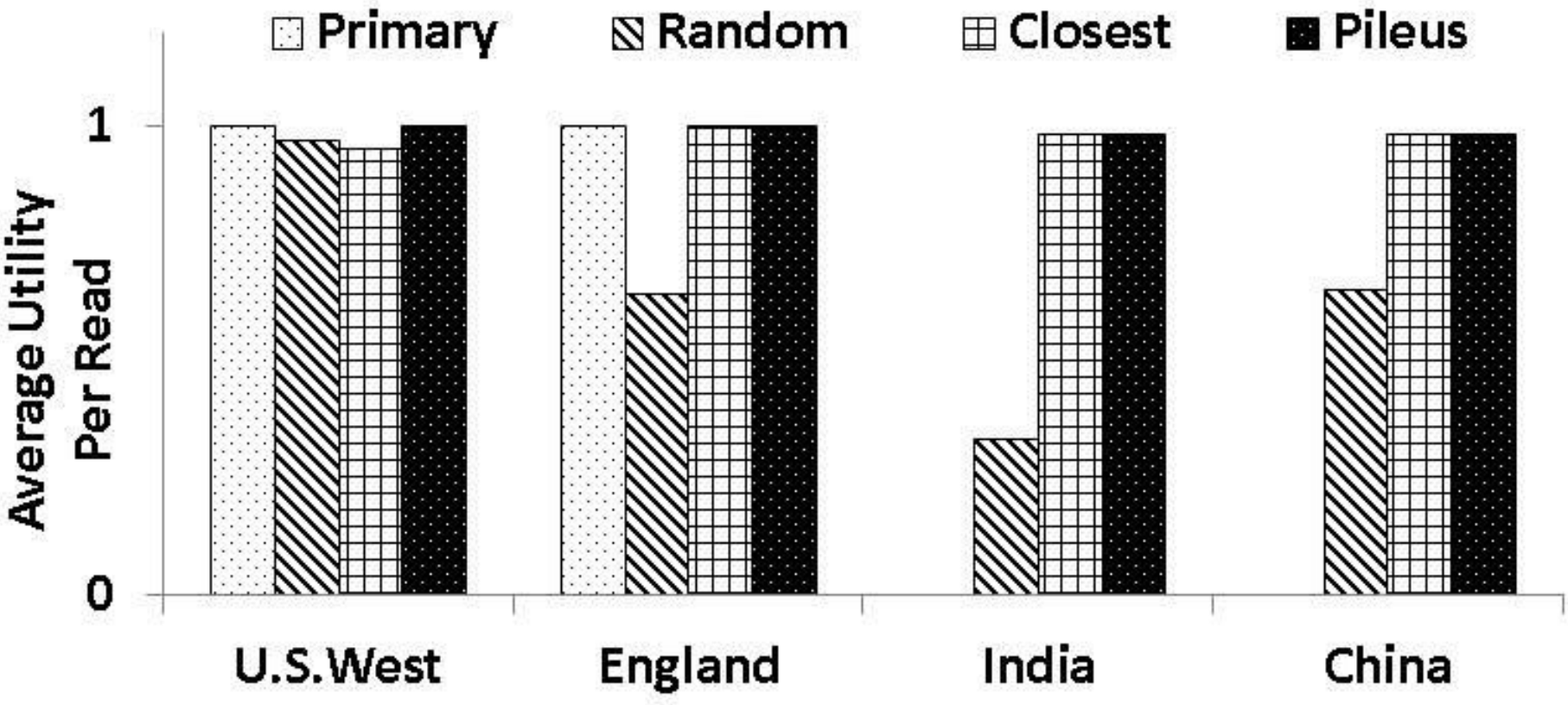
- Picked SLA

Rank	Consistency	Latency	Utility
1.	read-my-writes	300ms	1.0
2.	eventual	300ms	0.5

- Network topology



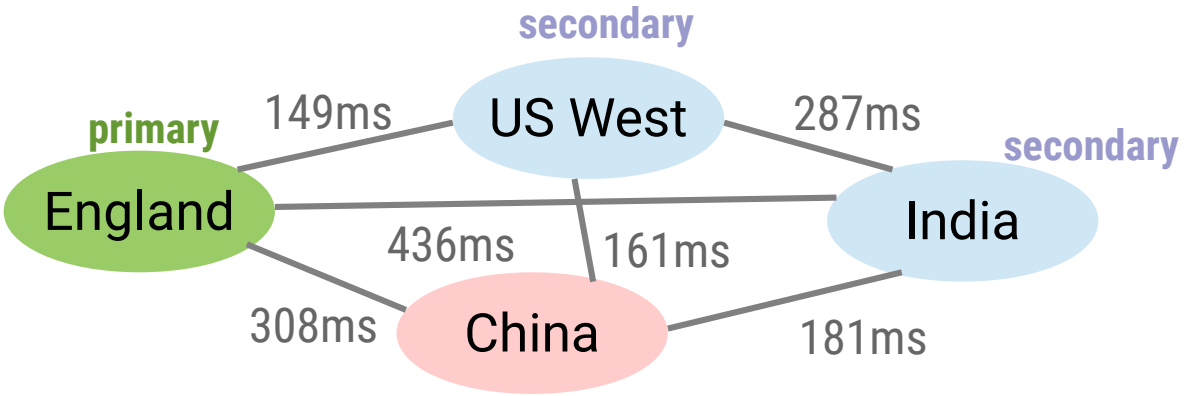
Experimental set-up: shopping cart



Experimental set-up: shopping cart

Client	*Target SubSLA*	*Get from U.S.*	*Get from England*	*Get from India*	*SubSLA Met*	*Avg. Utility*
U.S.	1	90.9%	9.1%	0%	100%	1.0
	2	0%	0%	0%	0%	
England	1	0%	100%	0%	100%	1.0
	2	0%	0%	0%	0%	
India	1	0.2%	0%	95.9%	96.1%	0.98
	2	0%	0%	3.9%	3.9%	
China	1	95.1%	0%	0.4%	95.5%	0.98
	2	4.5%	0%	0%	4.5%	

Consistency	Latency	Utility
read-my-writes	300ms	1.0
eventual	300ms	0.5

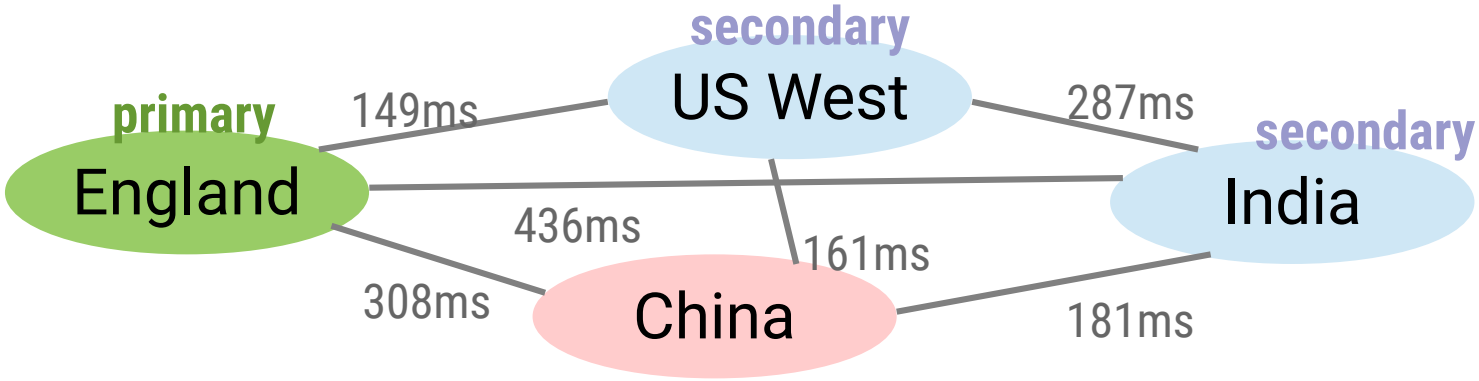


Experimental set-up: password

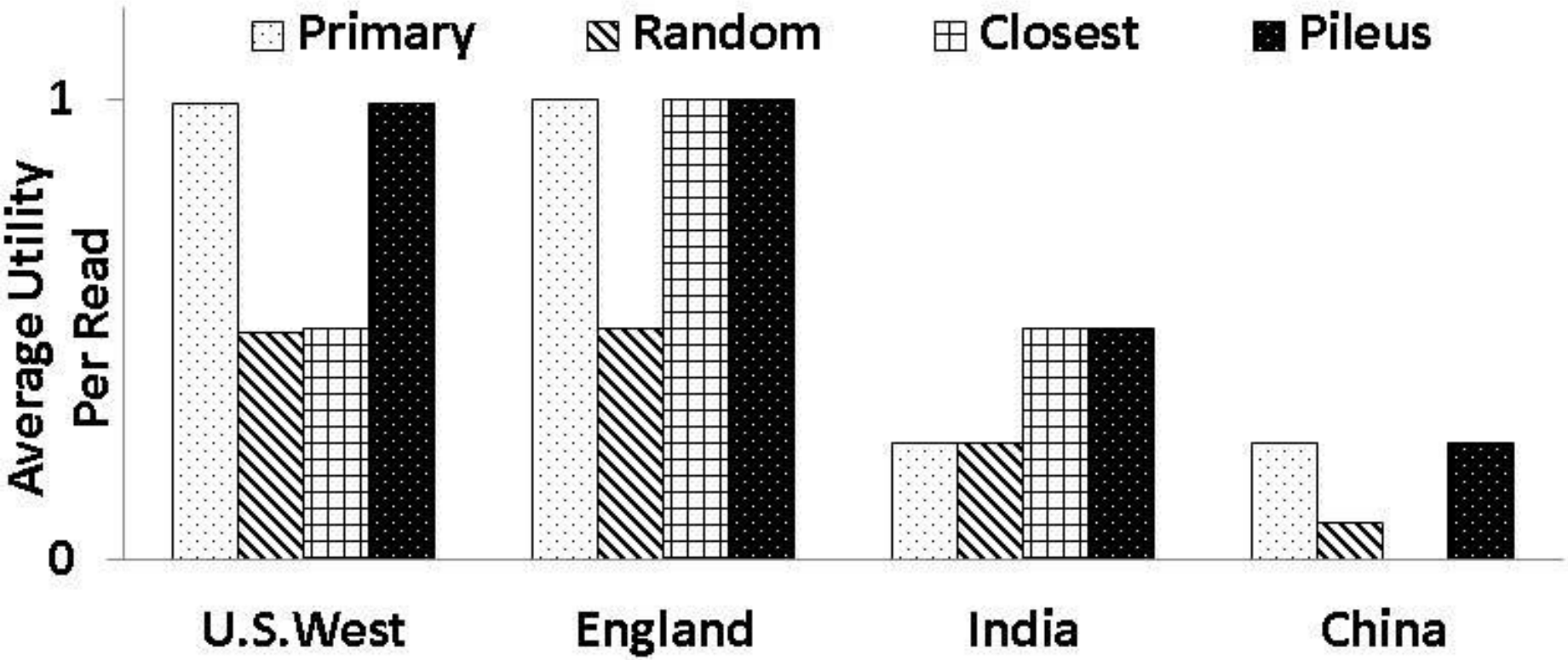
- Picked SLA

Rank	Consistency	Latency	Utility
1.	strong	150ms	1.0
2.	eventual	150ms	0.5
3.	strong	1s	0.25

- Network topology – the same as before



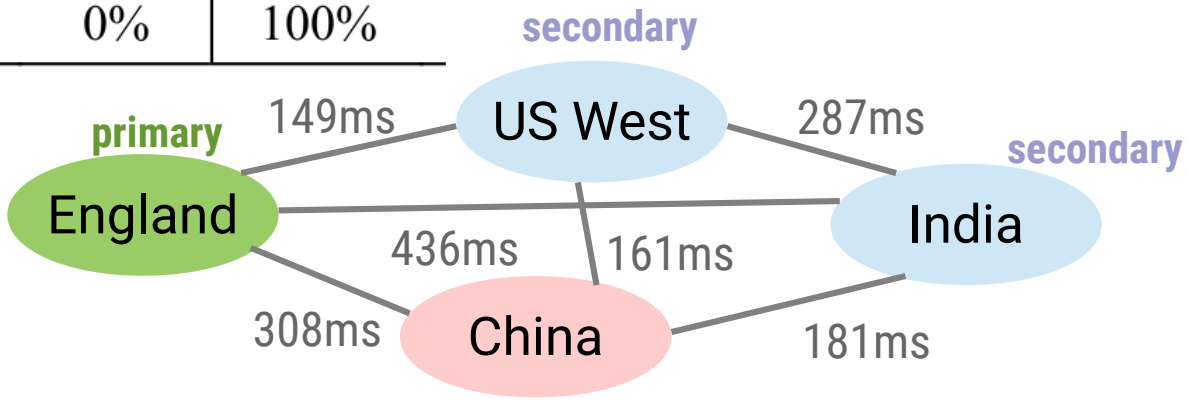
Experimental set-up: password



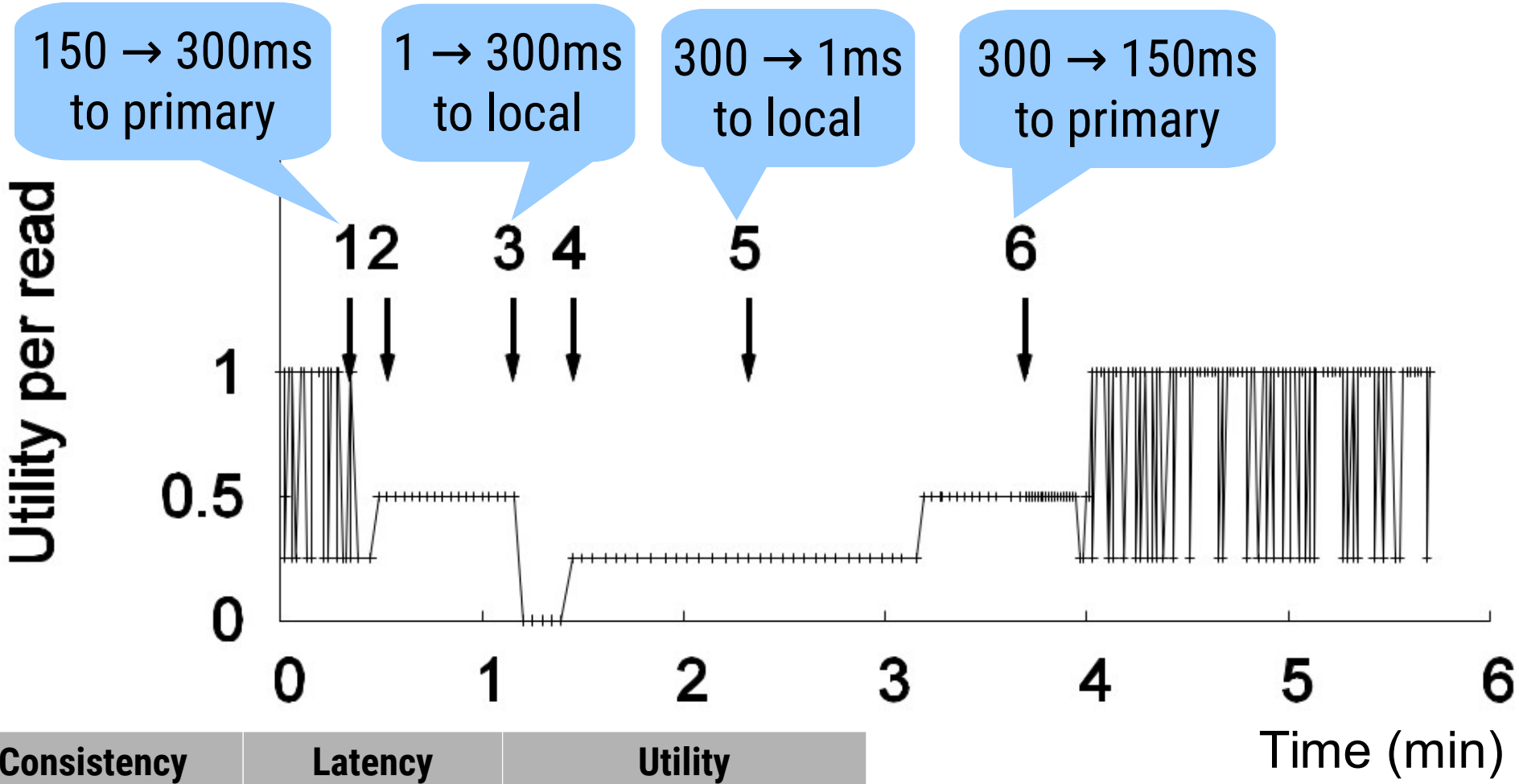
Experimental set-up: password

Client	*Target SubSLA*	*Get from U.S.*	*Get from England*	*Get from India*	*SubSLA Met*	*Avg. Utility*
U.S.	1	0%	100%	0%	99.4%	0.99
	2	0%	0%	0%	0%	
	3	0%	0%	0%	0.6%	
England	1	0%	100%	0%	100%	1.0
	2	0%	0%	0%	0%	
	3	0%	0%	0%	0%	
India	1	0%	0%	0%	0%	0.5
	2	0%	0%	100%	100%	
	3	0%	0%	0%	0%	
China	1	0%	0%	0%	0%	0.25
	2	0%	0%	0%	0%	
	3	0%	100%	0%	100%	

Consistency	Latency
strong	150ms
eventual	150ms
strong	1s

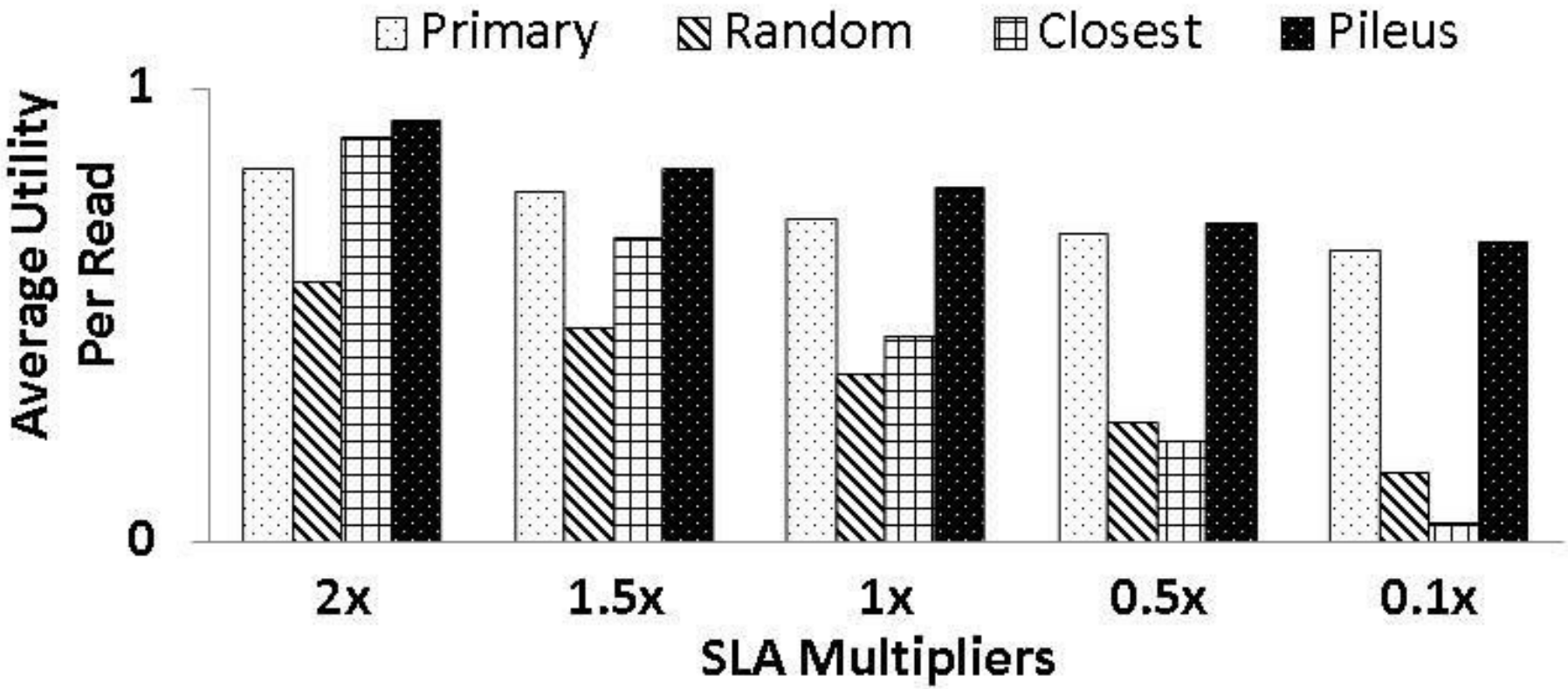


Varying latency



Consistency	Latency	Utility
strong	150ms	1.0
eventual	150ms	0.5
strong	1s	0.25

Varying utility



Consistency	Latency	Utility
strong	150ms	1.0
eventual	150ms	0.5 × **multiplier**
strong	1s	0.25 × **multiplier**