BOLT-ON CAUSAL CONSISTENCY

Peter Bailis, Ali Ghodsi,
Joseph M. Hellerstein, Ion Stoica
UC Berkeley

SLIDES FROM SIGMOD 2013

PAPER AT

HTTP://BAILIS.ORG/PAPERS/BOLTON-SIGMOD2013.PDF

PBAILIS@CS.BERKELEY.EDU

July 2000:
CAP
Conjecture

July 2000: CAP *Conjecture*

A system facing network partitions must choose between either availability or strong consistency

July 2000:

CAP ~~Conjecture~~ Theorem

A system facing network partitions must choose between either availability or strong consistency

NoSQL

NoSQL

NoSQL

NoSQL

NoSQL

NoSQL

Strong consistency is out!

"Partitions matter, and so does low latency"

[cf. Abadi: PACELC]

...offer eventual consistency instead

Eventual Consistency

Extremely weak consistency model:
eventually all replicas agree on the same value

Eventual Consistency

Extremely weak consistency model:
eventually all replicas agree on the same value

Any value can be returned at any given time
…as long as it's eventually the same everywhere

Eventual Consistency

Extremely weak consistency model:
eventually all replicas agree on the same value

Any value can be returned at any given time
…as long as it's eventually the same everywhere

Provides liveness but no safety guarantees

Liveness: something good eventually happens
Safety: nothing bad ever happens

Do we have to give up safety if we want availability?

Do we have to give up safety if we want availability?

Do we have to give up safety if we want availability?

No! There's a **spectrum** of models.

Do we have to give up safety if we want availability?

No! There's a **spectrum** of models.

Consistency, Availability, and Convergence

Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin
The University of Texas at Austin

Abstract

We examine the limits of consistency in highly available and fault-tolerant distributed storage systems. We introduce a new property—*convergence*—to explore the these limits in a useful manner. Like consistency and availability, convergence formalizes a fundamental requirement of a storage system: writes by one correct node must eventually become observable to other connected correct nodes. Using convergence as our driving force, we make two additional contributions. First, we close the gap between what is known to be impossible (i.e. the consistency, availability, and partition-tolerance theorem) and known systems that are highly-available but that provide weaker consistency such as causal. Specifically, in an asynchronous system, we show that *natural causal* consistency, a strengthening of causal consistency that respects the real-time ordering of operations, provides a tight bound on consistency semantics that can be enforced without compromising availability and convergence. In an asynchronous system with Byzantine-failures, we show that it is impossible to implement many of the recently introduced *forking*-based consistency semantics without sacrificing either availability or convergence. Finally, we show that it is not necessary to compromise availability or convergence by showing that there exist practically useful semantics that are enforceable by available, convergent, and Byzantine-fault tolerant systems.

1 Introduction

This paper examines the limits of consistency in highly available and fault-tolerant distributed storage systems. The tradeoffs between consistency and availability [6, 24, 38] have been widely used in guiding system design. The consistency, availability, partition-tolerance (CAP) theorem [24] is often cited as the reason why systems designed for high availability, such as Dynamo [19] and Cassandra [13], choose to enforce the very weak *eventual consistency* [56] semantics [13, 19, 56]. Conversely, the CAP theorem has guided designers

Do we have to give up safety if we want availability?

No! There's a **spectrum** of models.

Consistency, Availability, and Convergence

Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin
The University of Texas at Austin

Abstract

We examine the limits of consistency in highly available and fault-tolerant distributed storage systems. We introduce a new property—*convergence*—to explore the these limits in a useful manner. Like consistency and availability, convergence formalizes a fundamental requirement of a storage system: writes by one correct node must eventually become observable to other connected correct nodes. Using convergence as our driving force, we make two additional contributions. First, we close the gap between what is known to be impossible (i.e. the consistency, availability, and partition-tolerance theorem) and known systems that are highly-available but that provide weaker consistency such as causal. Specifically, in an asynchronous system, we show that *natural causal* consistency, a strengthening of causal consistency that respects the real-time ordering of operations, provides a tight bound on consistency semantics that can be enforced without compromising availability and convergence. In an asynchronous system with Byzantine-failures, we show that it is impossible to implement many of the recently introduced *forking*-based consistency semantics without sacrificing either availability or convergence. Finally, we show that it is not necessary to compromise availability or convergence by showing that there exist practically useful semantics that are enforceable by available, convergent, and Byzantine-fault tolerant systems.

1 Introduction

This paper examines the limits of consistency in highly available and fault-tolerant distributed storage systems. The tradeoffs between consistency and availability [6, 24, 38] have been widely used in guiding system design. The consistency, availability, partition-tolerance (CAP) theorem [24] is often cited as the reason why systems designed for high availability, such as Dynamo [19] and Cassandra [13], choose to enforce the very weak *eventual consistency* [56] semantics [13, 19, 56]. Conversely, the CAP theorem has guided designers

UT Austin TR:
No model stronger than Causal Consistency is achievable with HA


AT&T M-Cell
Home
Tweet
In reply to (null)
Cliff Moon
@moonpolysoft
@strlen really? I thought southbay was all about office parks and strip malls.
3 hours ago via web
Replies


AT&T M-Cell
Home
Tweet
In reply to (null)
Cliff Moon
@moonpolysoft
@strlen really? I thought southbay was all about office parks and strip malls.
3 hours ago via web
Replies

Why Causal Consistency?

Highly available, low latency operation

[UT Austin 2011 TR]

Long-identified useful "session" model

Natural fit for many modern apps

[Bayou Project, 1994-98]

Dilemma!

Eventual consistency is the
lowest common denominator across systems...

Dilemma!

Eventual consistency is the **lowest common denominator** across systems...

...yet eventual consistency is often **insufficient** for many applications...

Dilemma!

Eventual consistency is the **lowest common denominator** across systems...

...yet eventual consistency is often **insufficient** for many applications...

...and no production-ready storage systems offer highly available causal consistency.

In this talk...

show how to upgrade existing
stores to provide HA causal consistency

In this talk...

show how to upgrade existing
stores to provide HA causal consistency

Approach: bolt on a narrow *shim layer*
to upgrade eventual consistency

In this talk...

show how to upgrade existing
stores to provide HA causal consistency

Approach: bolt on a narrow *shim layer*
to upgrade eventual consistency

Outcome: architecturally separate safety
and liveness properties

Separation of Concerns

Separation of Concerns

Shim handles:

Consistency/visibility

Separation of Concerns

Shim handles:
Consistency/visibility

Consistency-related Safety
Mostly algorithmic
Small code base

Separation of Concerns

Shim handles:
Consistency/visibility

Consistency-related Safety
Mostly algorithmic
Small code base

Underlying store handles:
Messaging/propagation
Durability/persistence
Failure-detection/handling

Separation of Concerns

Shim handles:
Consistency/visibility

Consistency-related Safety
Mostly algorithmic
Small code base

Underlying store handles:
Messaging/propagation
Durability/persistence
Failure-detection/handling

Liveness and Replication
Lots of engineering
Reuse existing efforts!

Separation of Concerns

Shim handles:
Consistency/visibility

Consistency-related Safety
Mostly algorithmic
Small code base

Underlying store handles:
Messaging/propagation
Durability/persistence
Failure-detection/handling

Liveness and Replication
Lots of engineering
Reuse existing efforts!

Guarantee same (useful) semantics across systems!
Allows portability, modularity, comparisons

Bolt-on Architecture

Bolt-on *shim layer* upgrades the semantics of an eventually consistent data store

Clients only communicate with shim

Shim communicates with one of many different eventually consistent stores (generic)

Bolt-on Architecture

Bolt-on *shim layer* upgrades the semantics of an eventually consistent data store

Clients only communicate with shim

Shim communicates with one of many different eventually consistent stores (generic)

Treat EC store as "storage manager" of distributed DBMS

Bolt-on Architecture

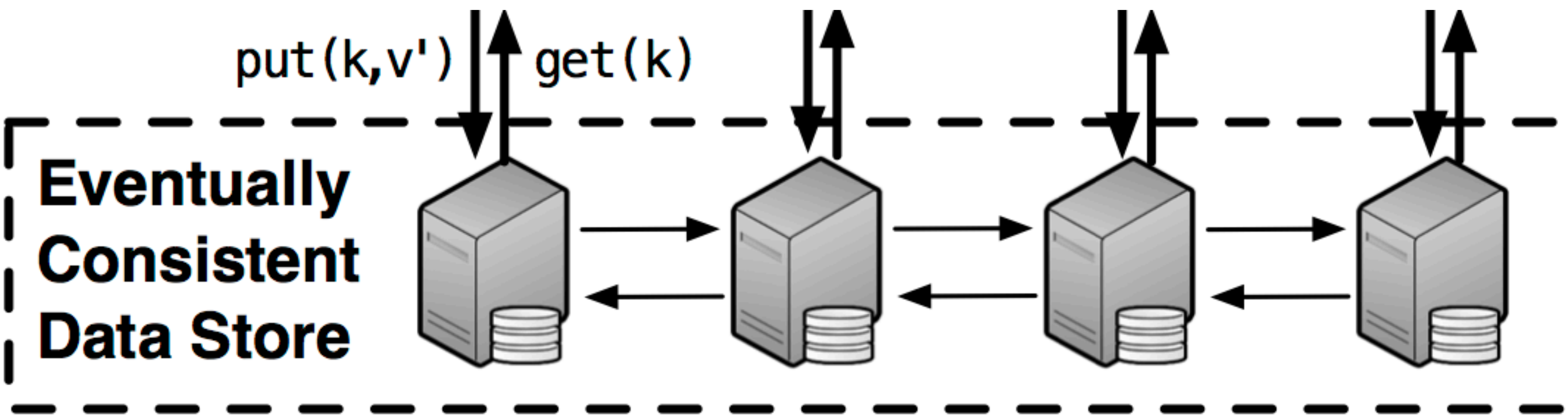
Bolt-on *shim layer* upgrades the semantics of an eventually consistent data store

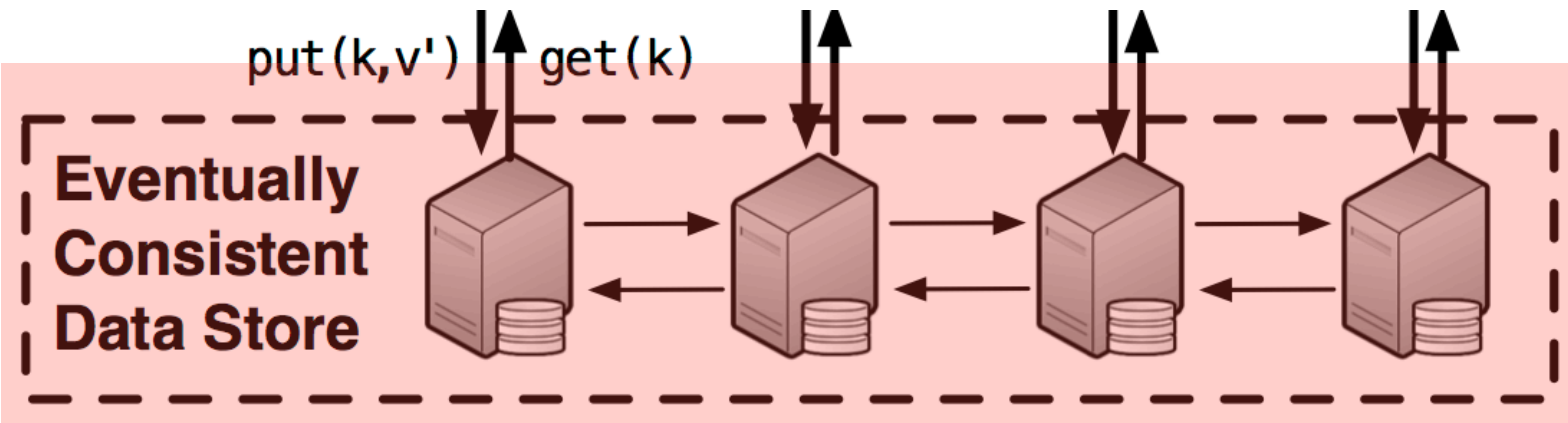
Clients only communicate with shim

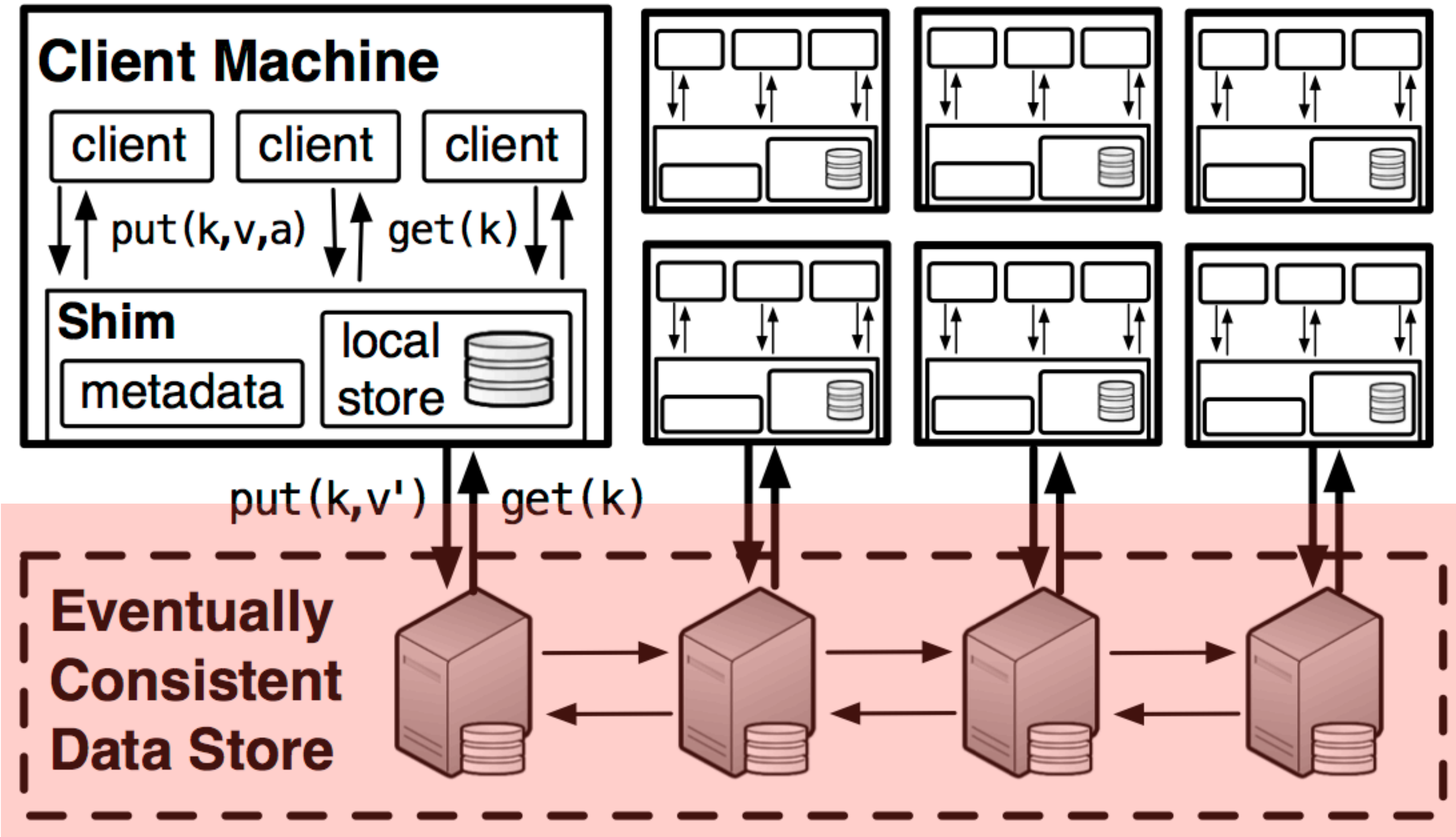
Shim communicates with one of many different eventually consistent stores (generic)

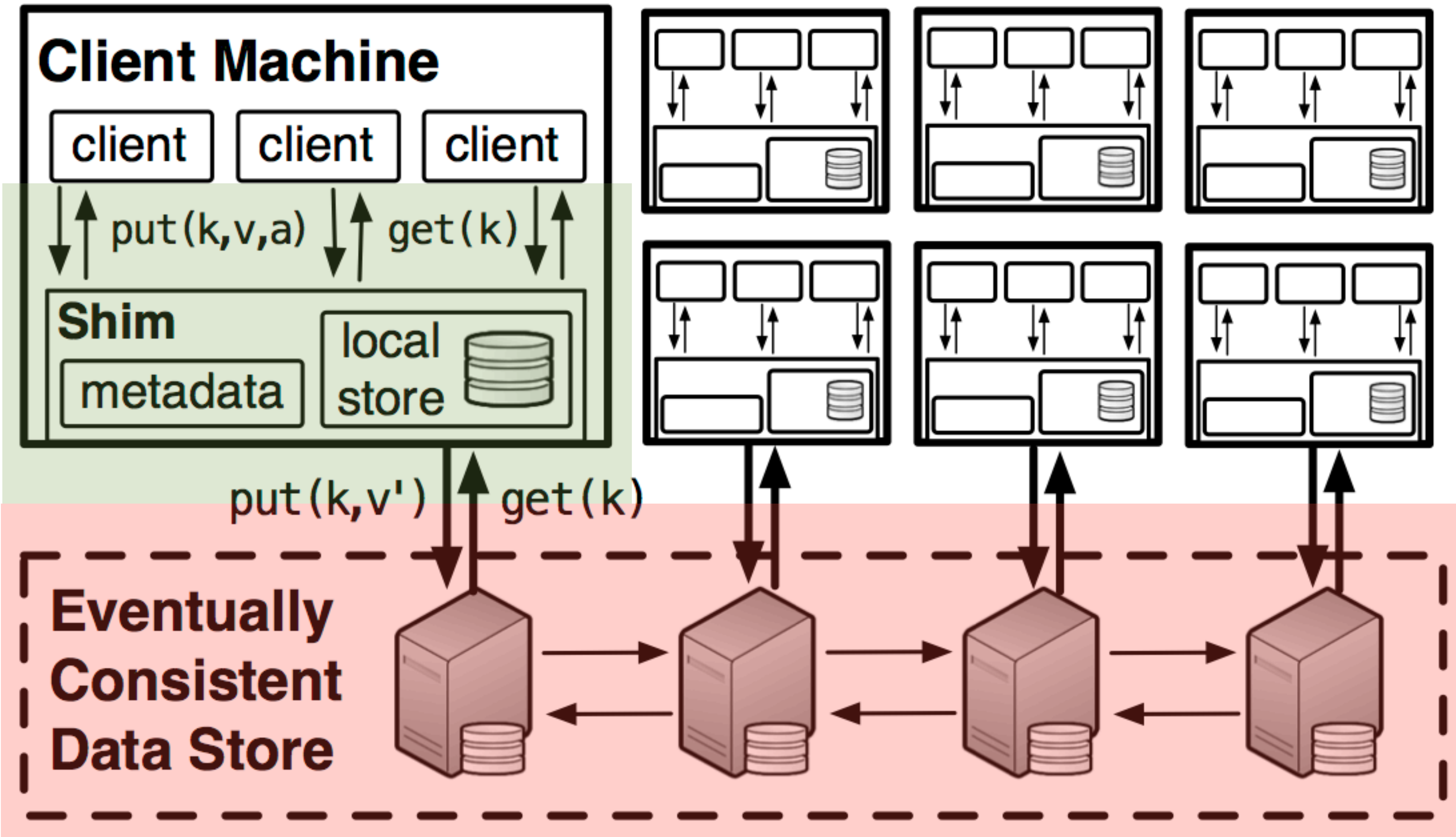
Treat EC store as "storage manager" of distributed DBMS

for now, an extreme: unmodified EC store

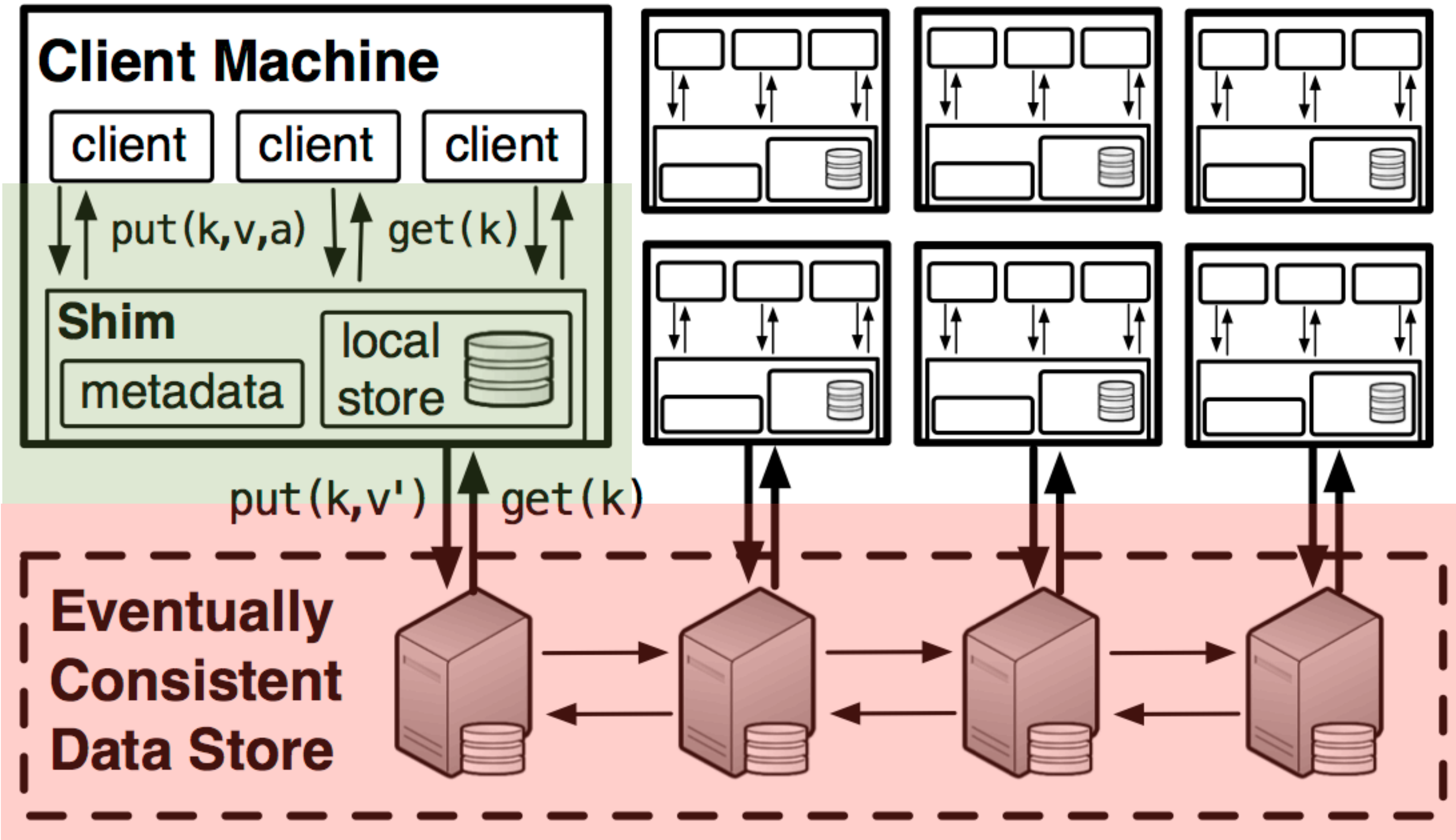

put(k,v')
get(k)
Eventually
Consistent
Data Store


put(k,v')
get(k)
Eventually
Consistent
Data Store


Client Machine
client
client
client
put(k,v,a)
get(k)
Shim
metadata
local store
put(k,v')
get(k)
Eventually Consistent Data Store


Client Machine
client
client
client
put(k,v,a)
get(k)
Shim
metadata
local
store
put(k,v')
get(k)
Eventually
Consistent
Data Store

BOLT-ON CAUSAL CONSISTENCY



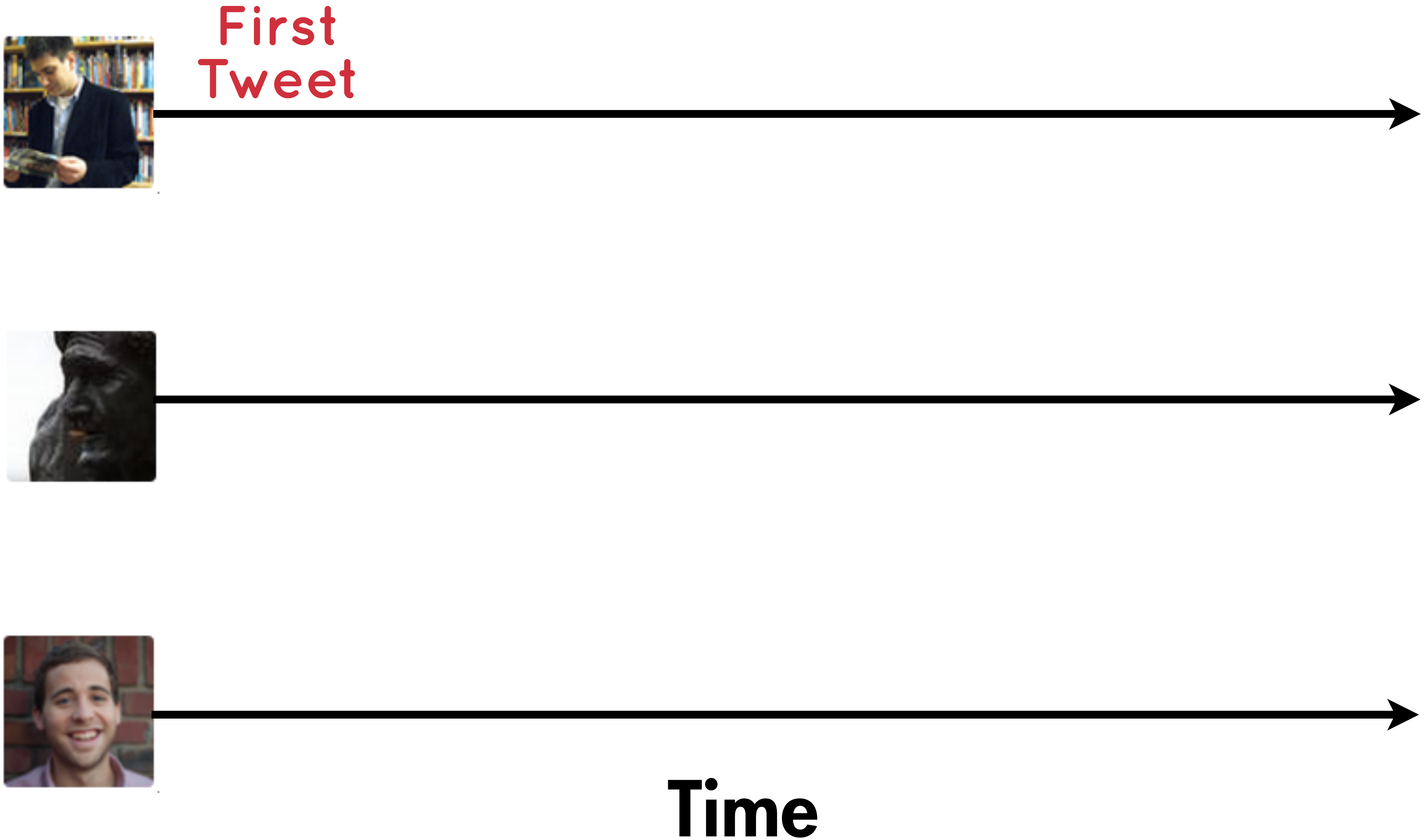

AT&T M-Cell
Home
Tweet
In reply to (null)
Cliff Moon
@moonpolysoft
@strlen really? I thought southbay was all about office parks and strip malls.
3 hours ago via web
Replies

What is Causal Consistency?

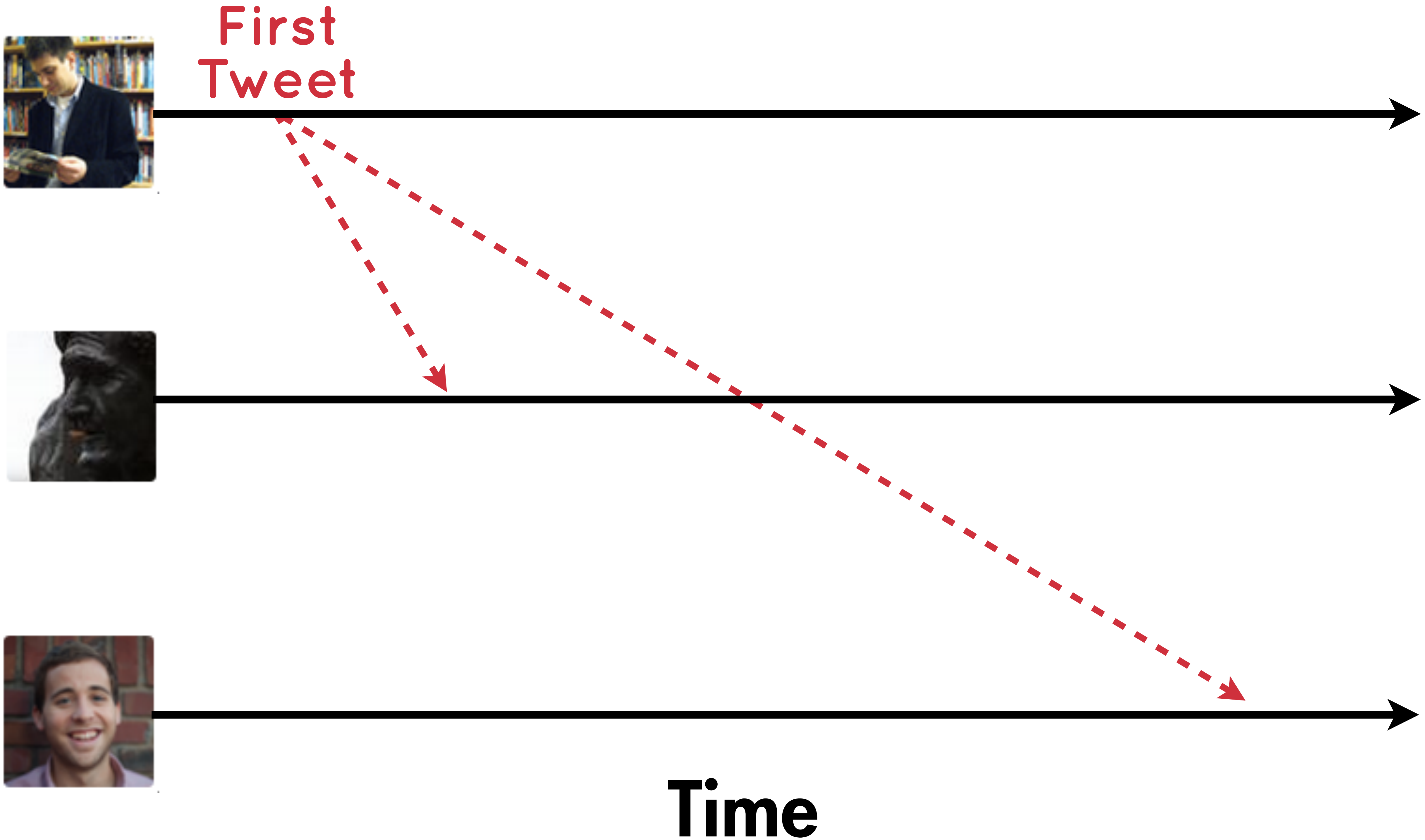
What is Causal Consistency?

What is Causal Consistency?

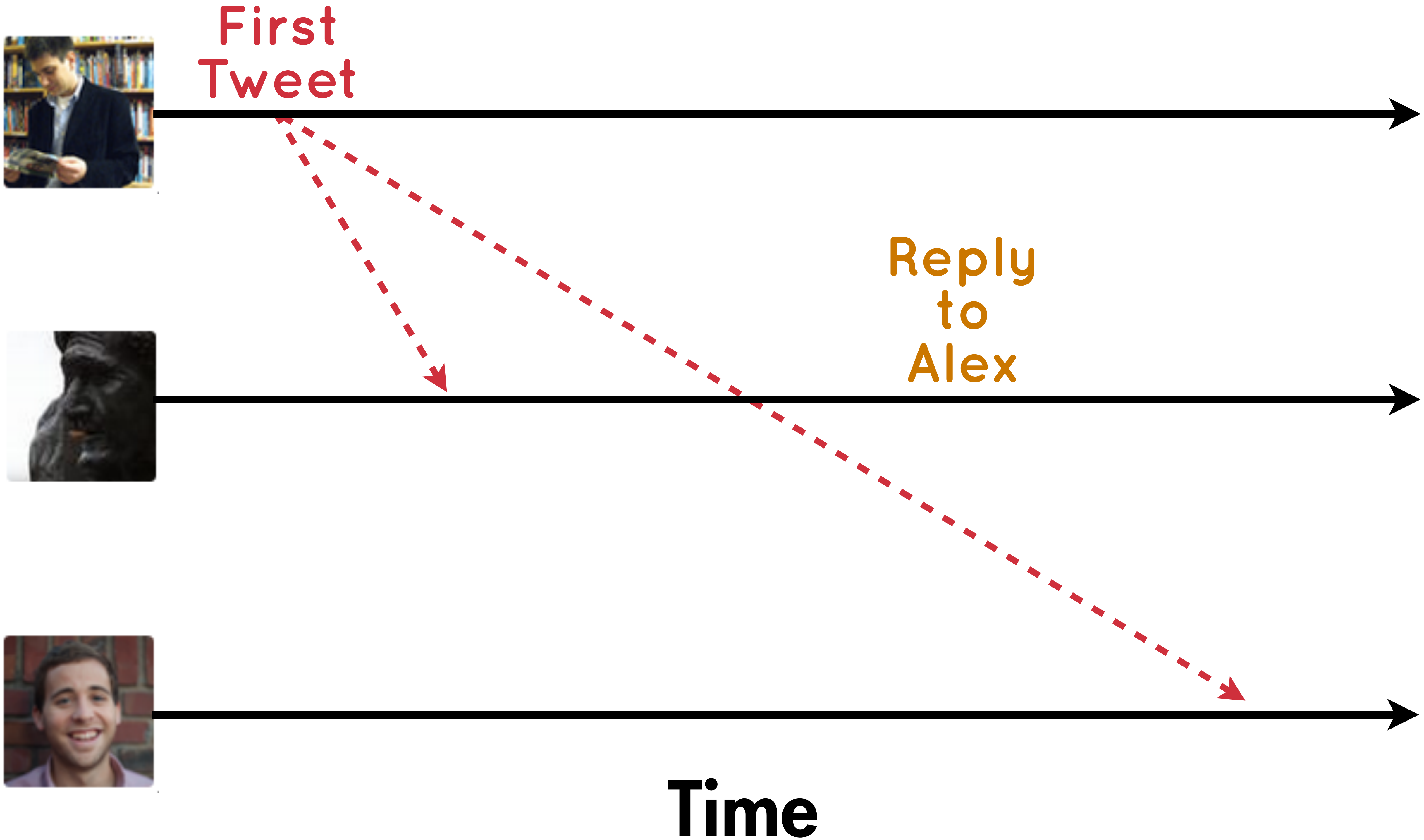
What is Causal Consistency?



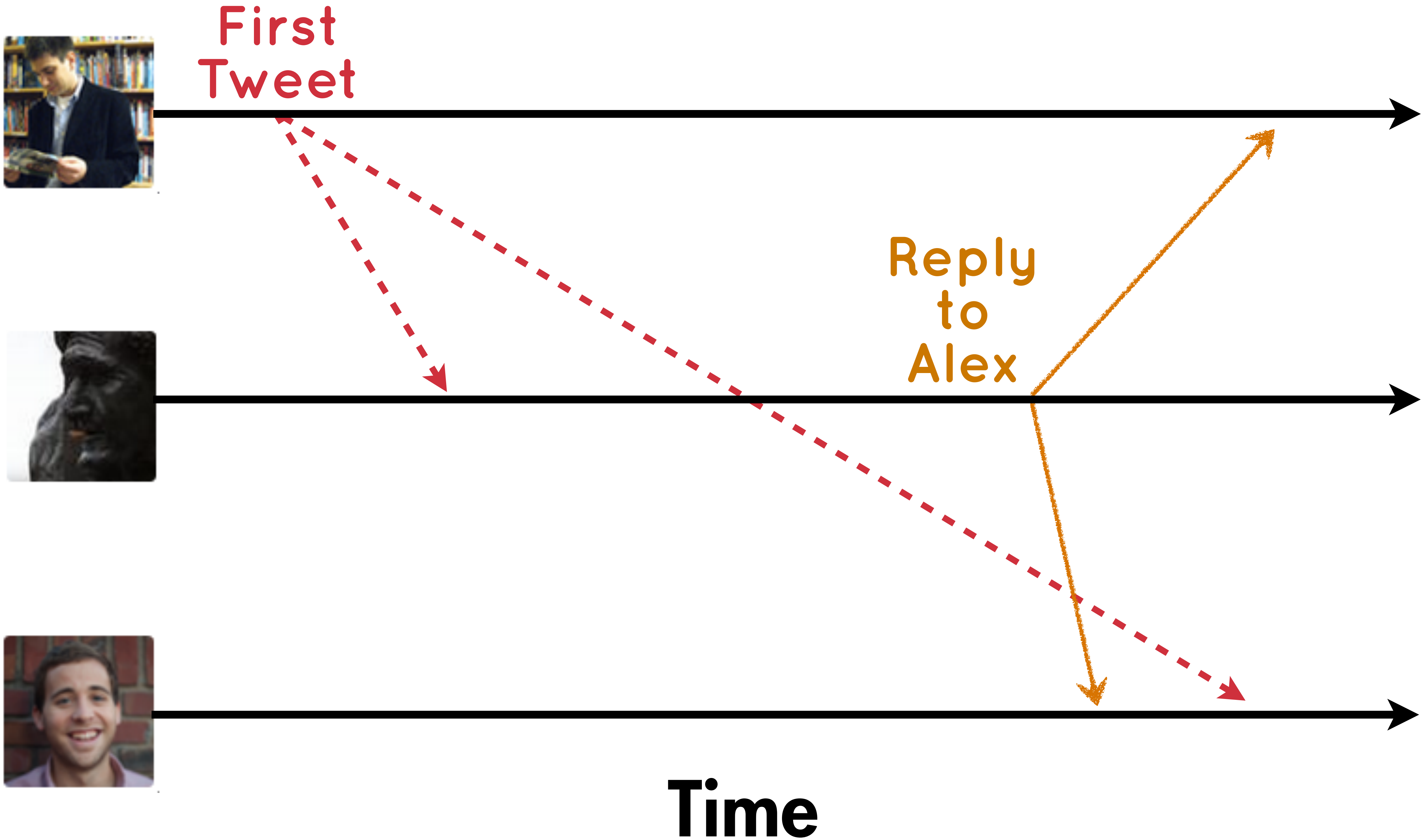
What is Causal Consistency?



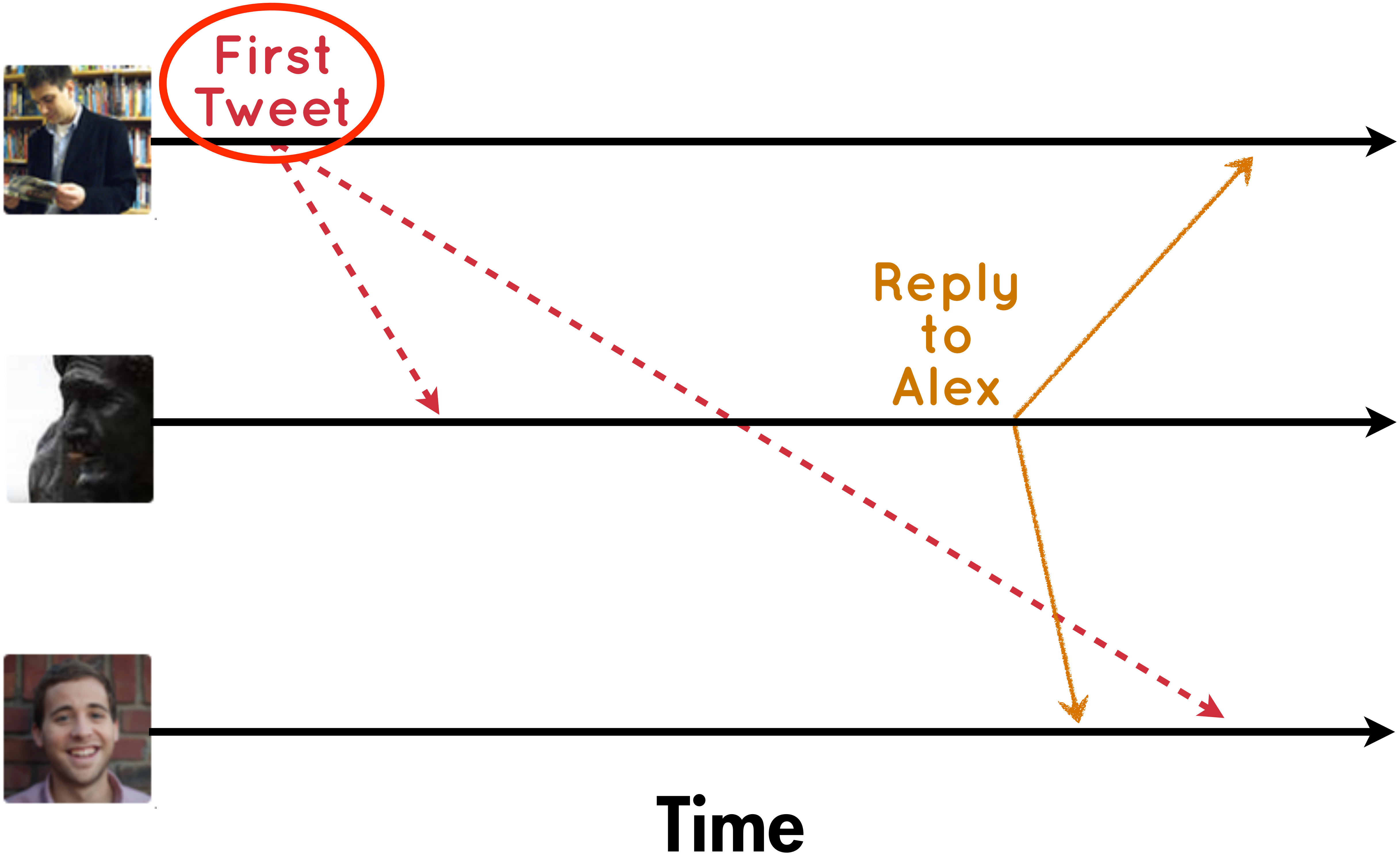
What is Causal Consistency?



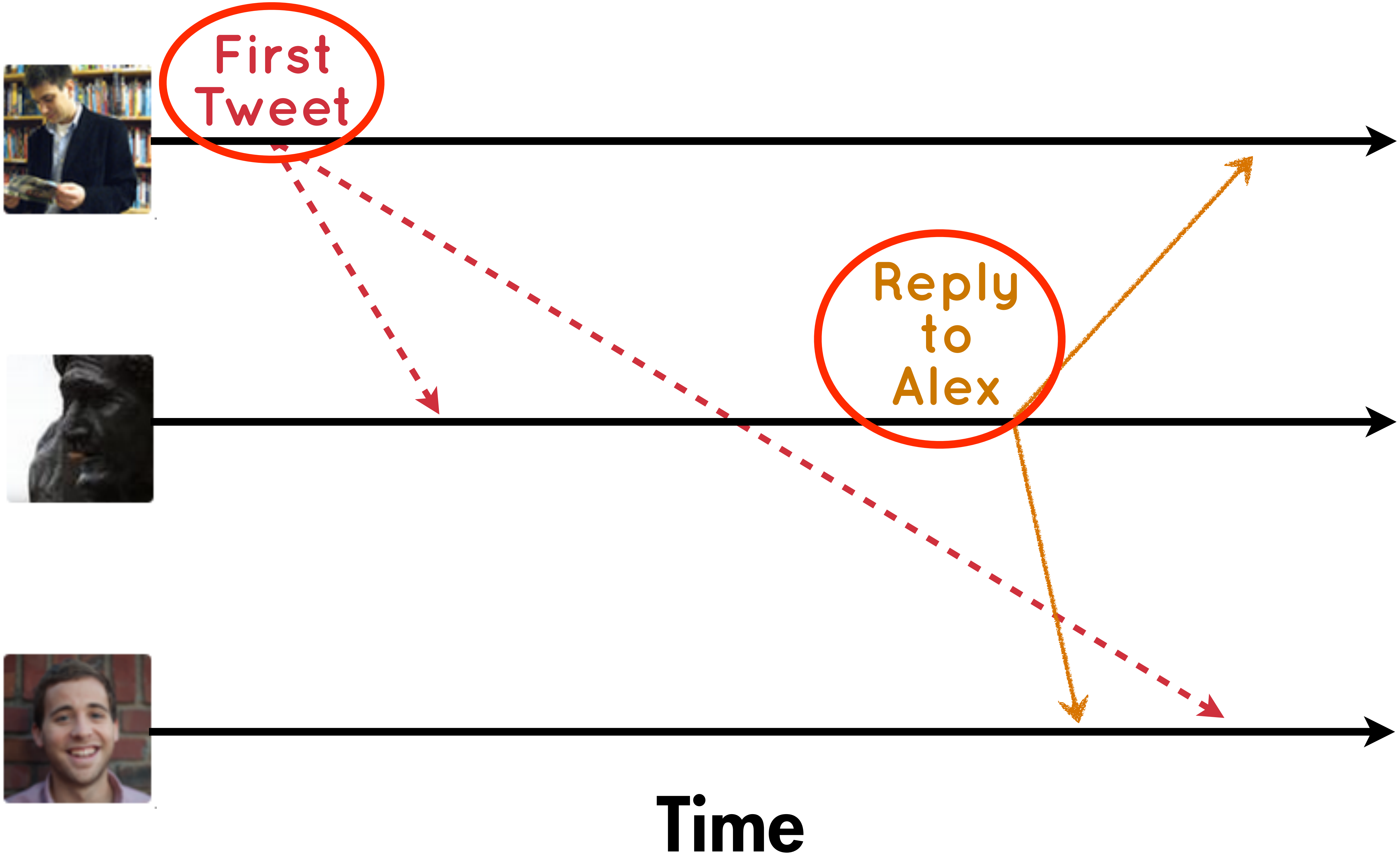
What is Causal Consistency?



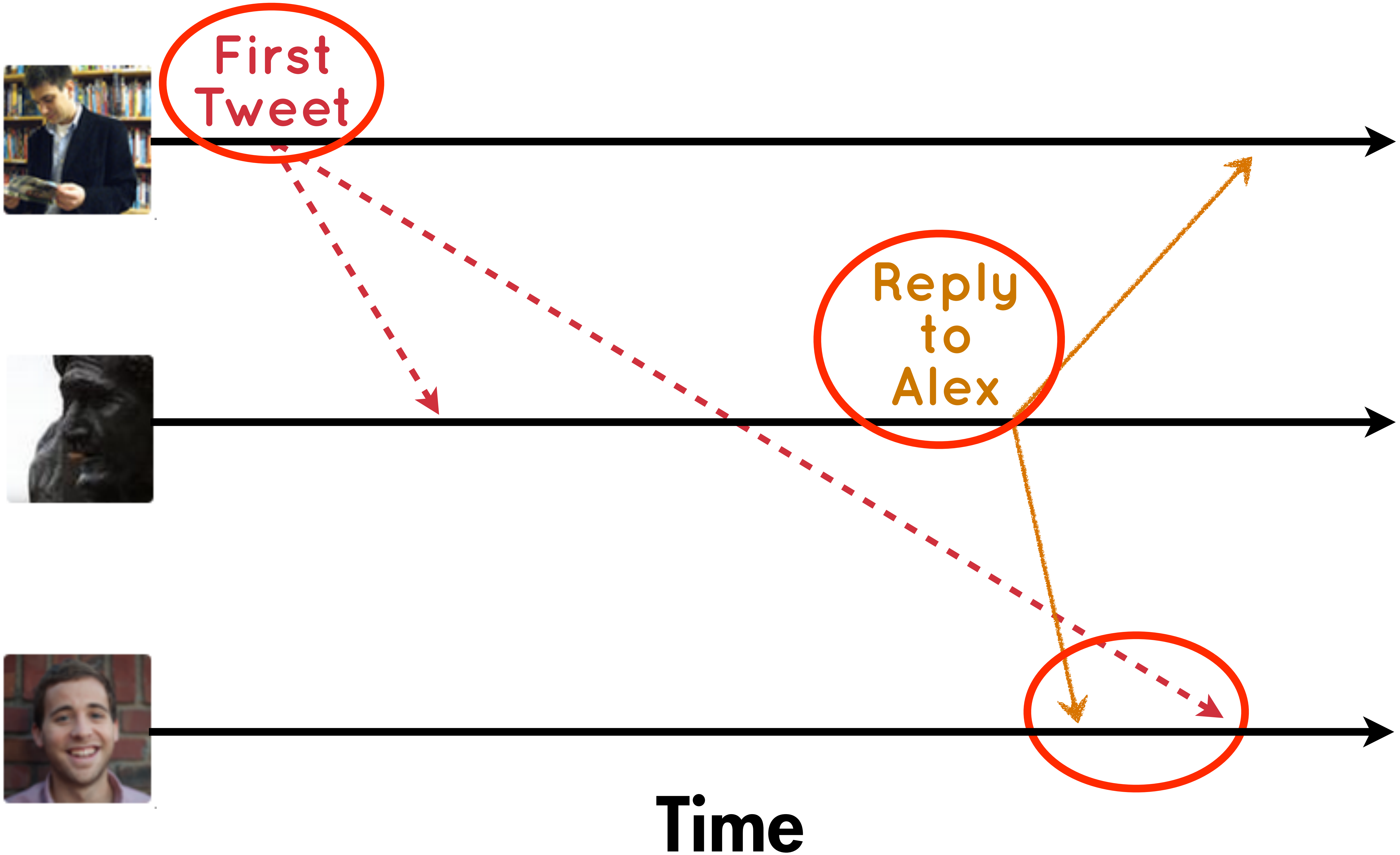
What is Causal Consistency?



What is Causal Consistency?



What is Causal Consistency?



What is Causal Consistency?

Reads obey:

1.) Writes Follow Reads
("happens-before")

2.) Program order

3.) Transitivity

[Lamport 1978]

What is Causal Consistency?

Reads obey:

1.) Writes Follow Reads
("happens-before")

2.) Program order

3.) Transitivity

[Lamport 1978]

Here, applications *explicitly* define happens-before for each write ("explicit causality")

[Ladin et al. 1990,
cf. Bailis et al. 2012]

What is Causal Consistency?

Reads obey:

1.) Writes Follow Reads ("happens-before")

2.) Program order

3.) Transitivity

[Lamport 1978]

Here, applications *explicitly* define happens-before for each write ("explicit causality")

[Ladin et al. 1990, cf. Bailis et al. 2012]

First Tweet

Reply to Alex

What is Causal Consistency?

Reads obey:

1.) Writes Follow Reads
("happens-before")

2.) Program order

3.) Transitivity

[Lamport 1978]

Here, applications *explicitly* define happens-before for each write ("explicit causality")

[Ladin et al. 1990, cf. Bailis et al. 2012]

First Tweet

happens-before →

Reply to Alex

Developers API Health Blog Discussions Documentation Search

POST statuses/update

View | What links here

Updated on Tue, 2012-11-20 08:24 **API version 1.1**

Updates the authenticating user's current status, also known as tweeting. To upload an image to accompany the tweet, use POST statuses/update_with_media.

For each update attempt, the update text is compared with the authenticating user's recent tweets. Any attempt that would result in duplication will be blocked, resulting in a 403 error. Therefore, a user cannot submit the same status twice in a row.

in_reply_to_status_id optional	The ID of an existing status that the update is in reply to. **Note:**: This parameter will be ignored unless the author of the tweet this parameter references is mentioned within the status text. Therefore, you must include `@username`, where `username` is the author of the referenced tweet, within the update.

https://dev.twitter.com/docs/api/1.1/post/statuses/update

Developers API Health Blog Discussions Documentation Search

POST statuses/update

View What links here

Updated on Tue, 2012-11-20 08:24 **API version 1.1**

Updates the authenticating user's current status, also known as tweeting. To upload an image to accompany the tweet, use POST statuses/update_with_media.

For each update attempt, the update text is compared with the authenticating user's recent tweets. Any attempt that would result in duplication will be blocked, resulting in a 403 error. Therefore, a user cannot submit the same status twice in a row.

in_reply_to_status_id optional	The ID of an existing status that the update is in reply to. **Note:**: This parameter will be ignored unless the author of the tweet this parameter references is mentioned within the status text. Therefore, you must include `@username`, where `username` is the author of the referenced tweet, within the update.

happens-before

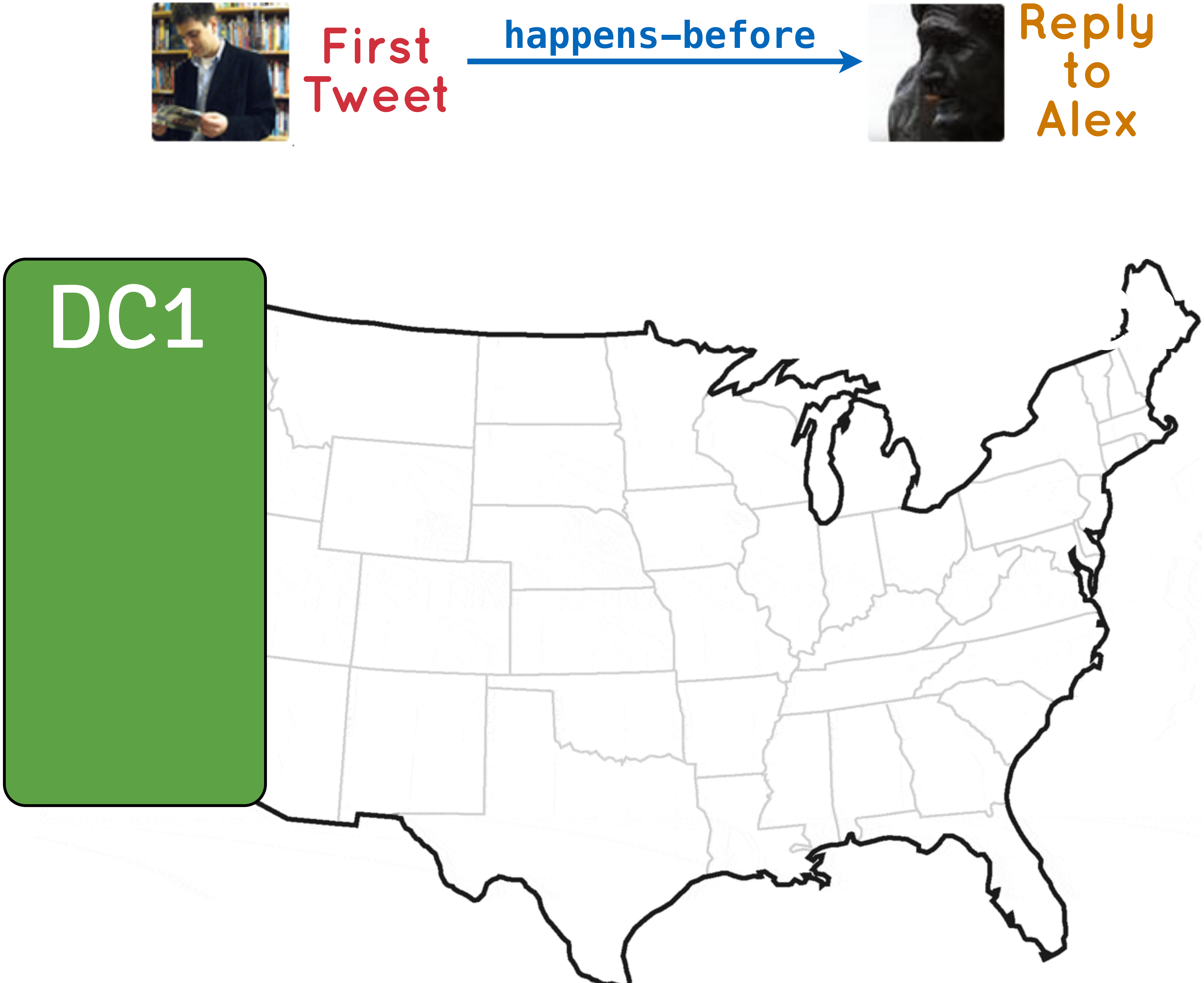
First Tweet —happens-before→ Reply to Alex

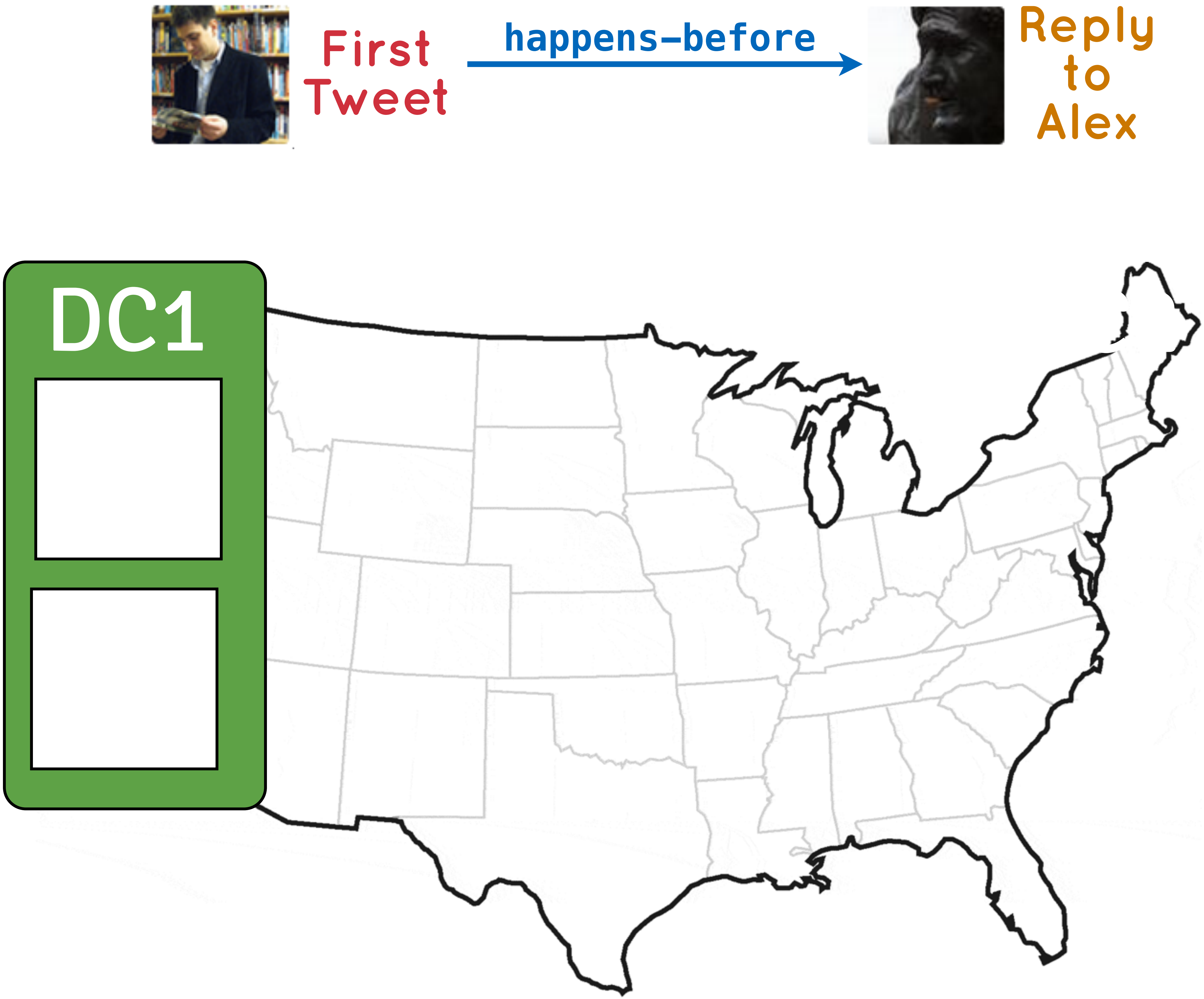
https://dev.twitter.com/docs/api/1.1/post/statuses/update

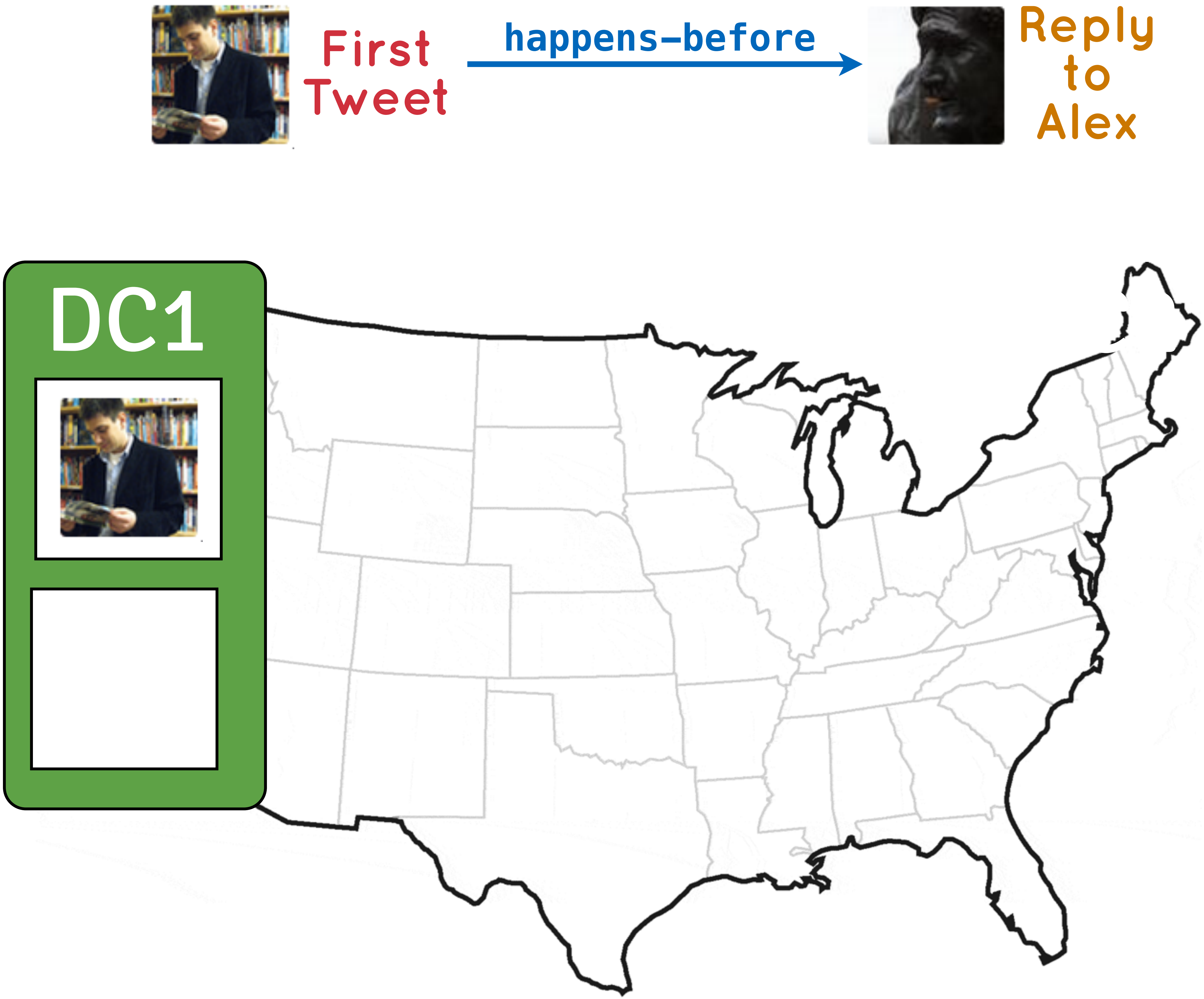
First Tweet

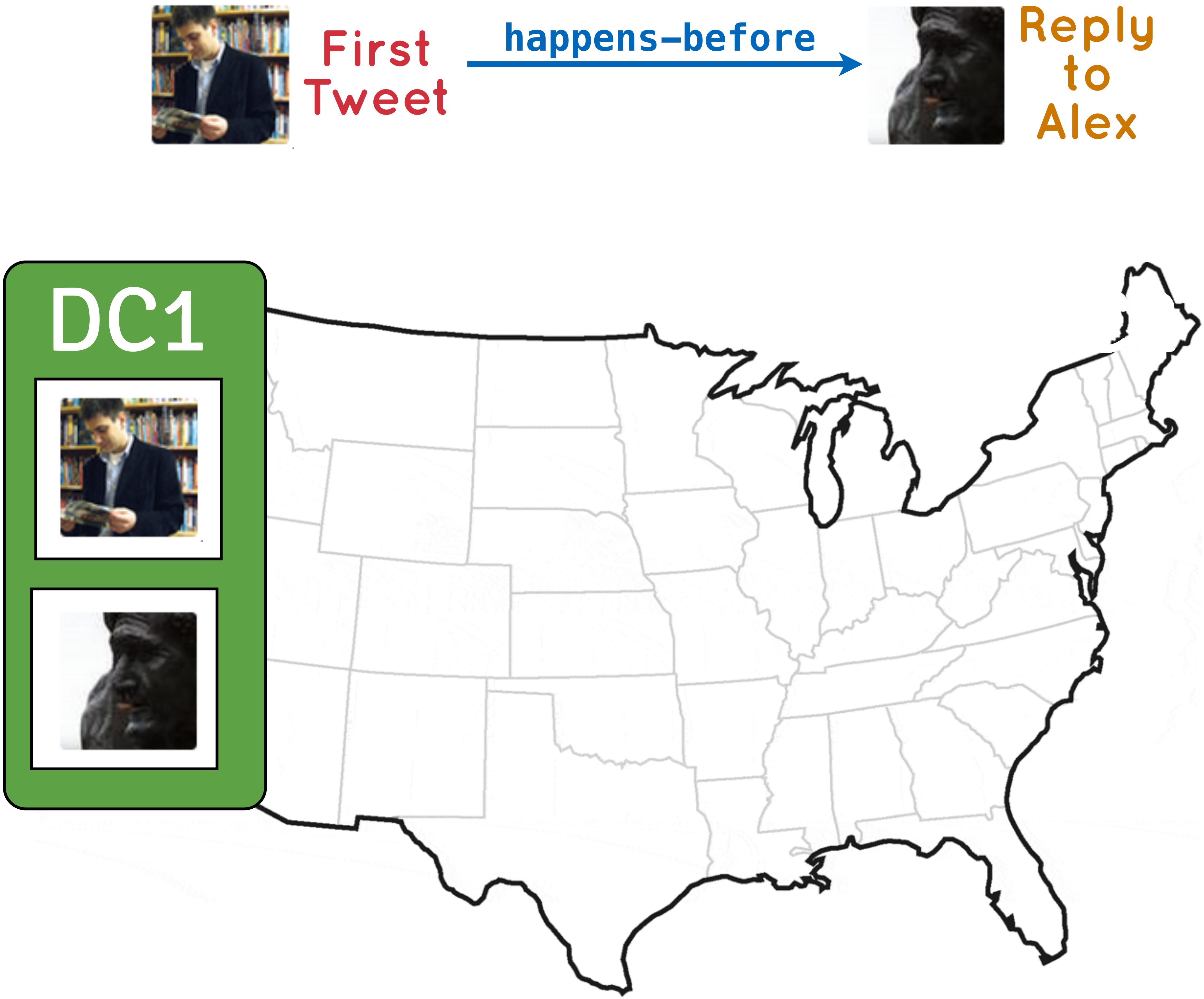
happens-before

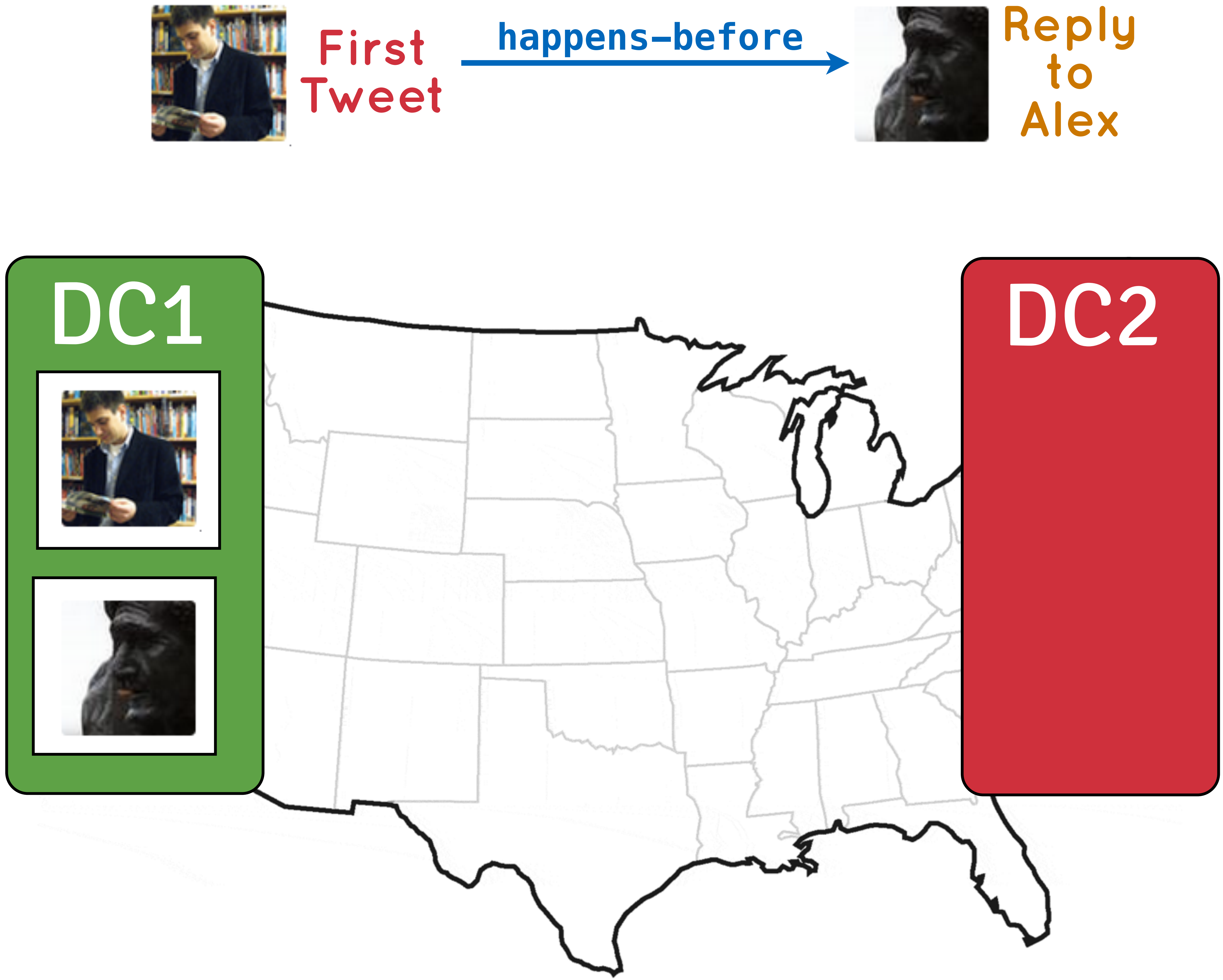
Reply to Alex


First Tweet
happens-before
Reply to Alex
DC1

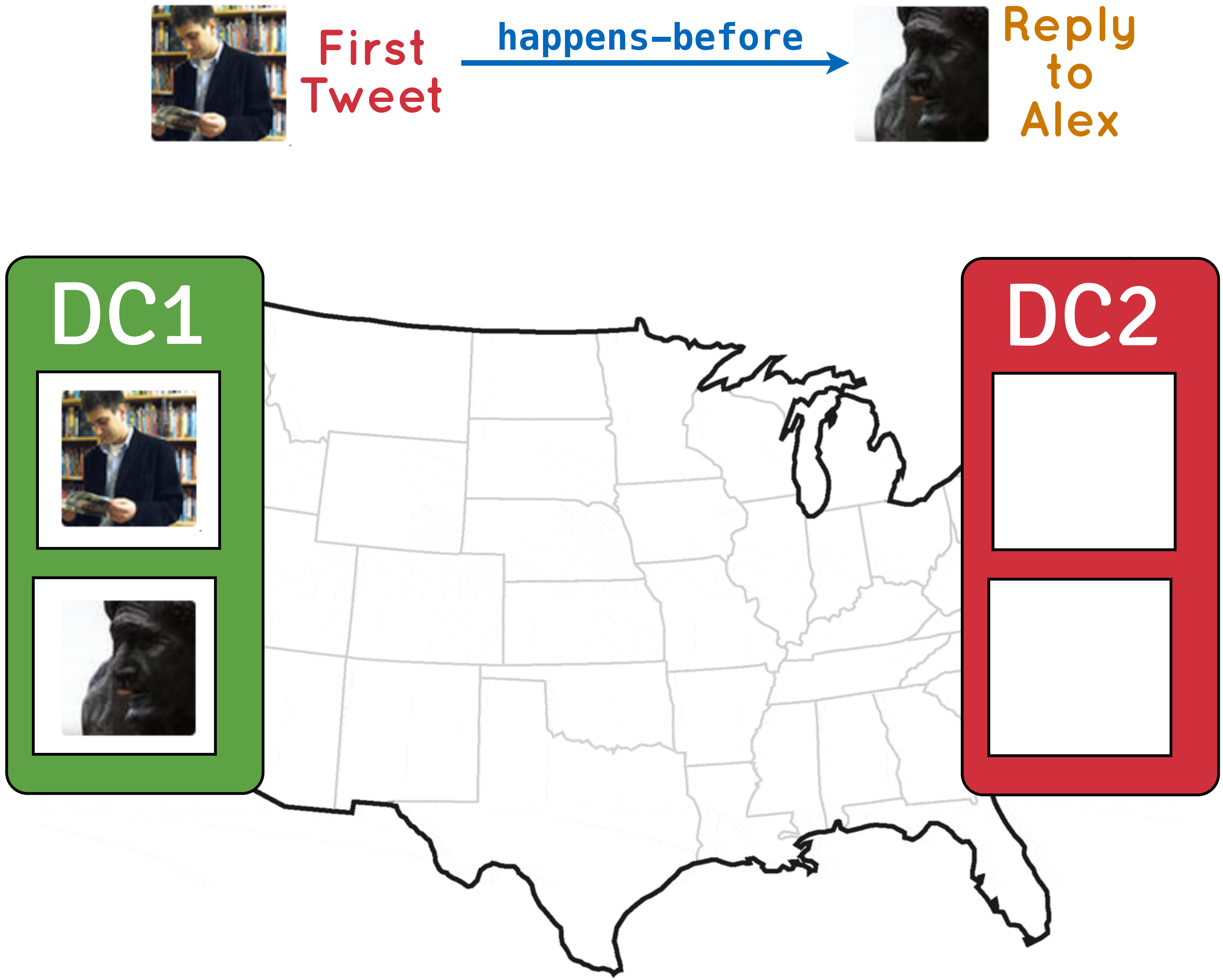

First Tweet
happens-before
Reply to Alex
DC1

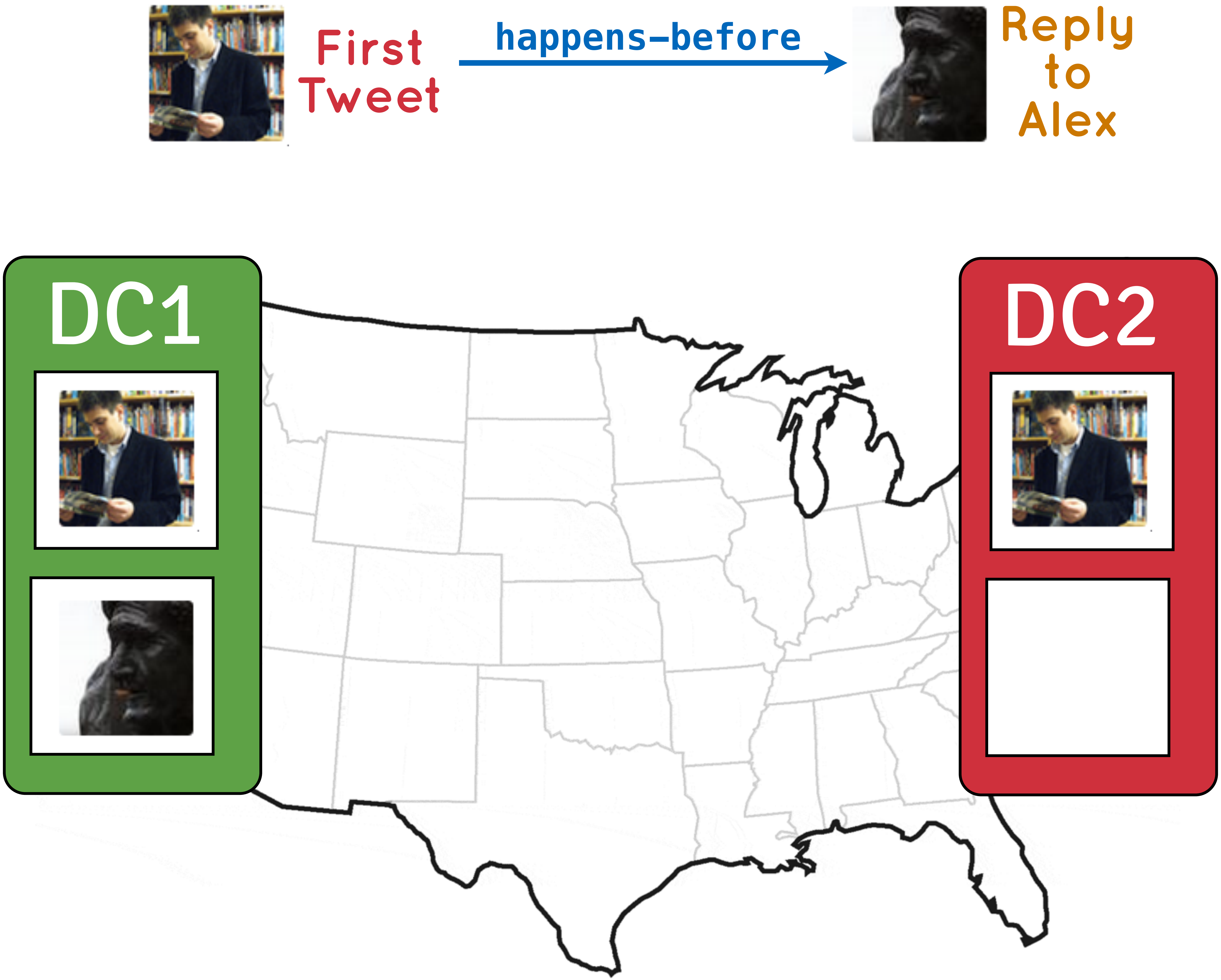

First Tweet
happens-before
Reply to Alex
DC1

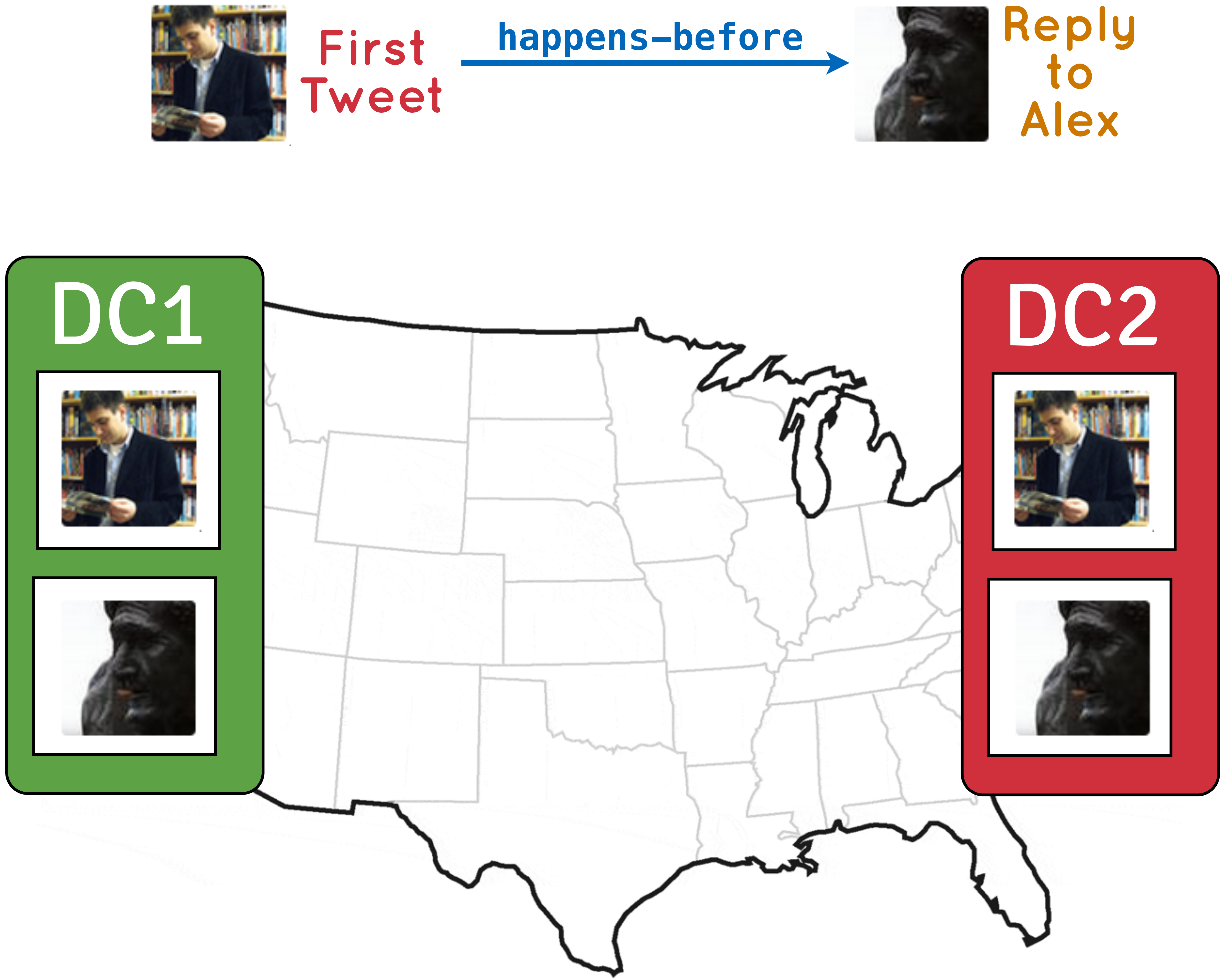

First Tweet
happens-before
Reply to Alex
DC1

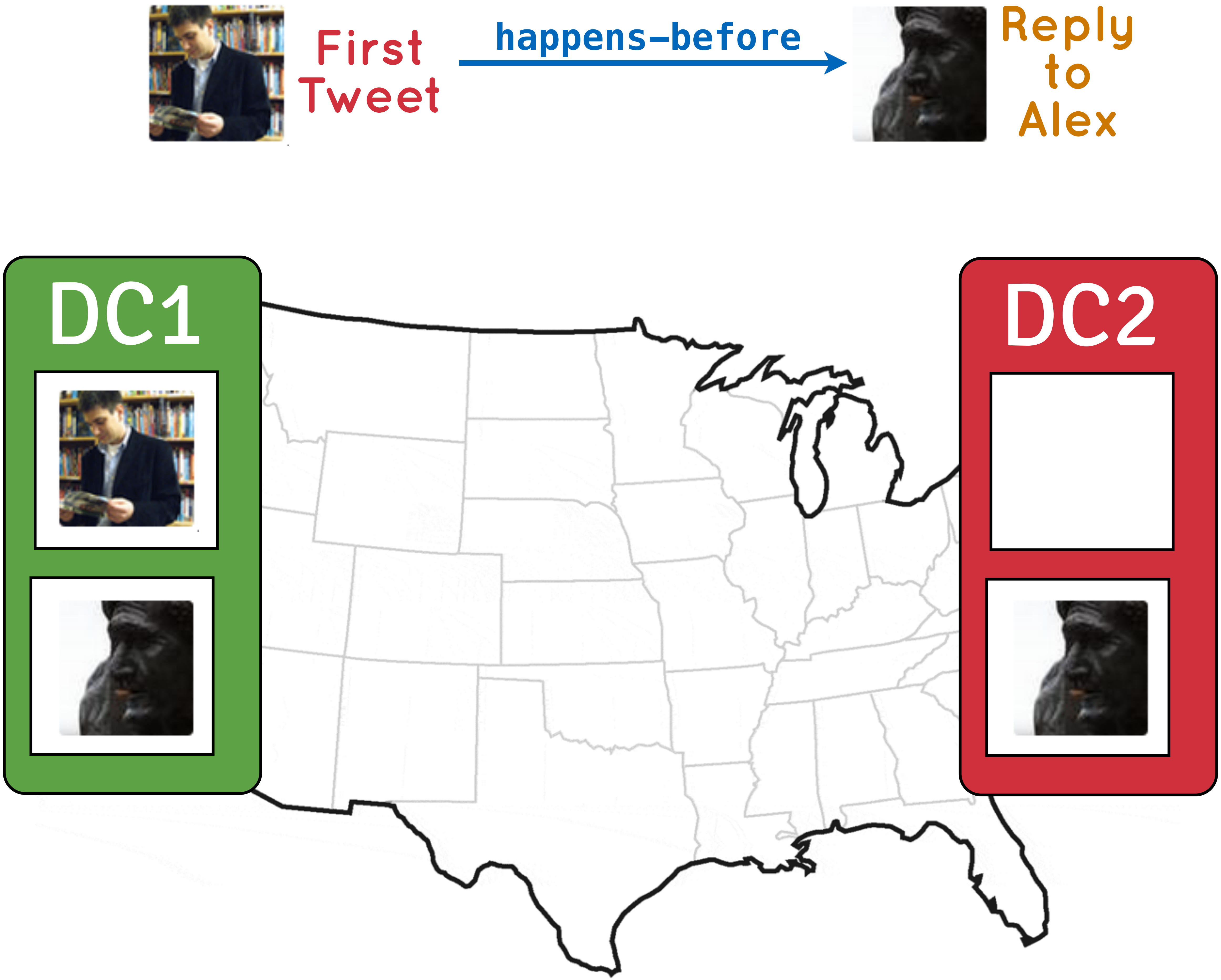


First Tweet
happens-before
Reply to Alex
DC1
DC2

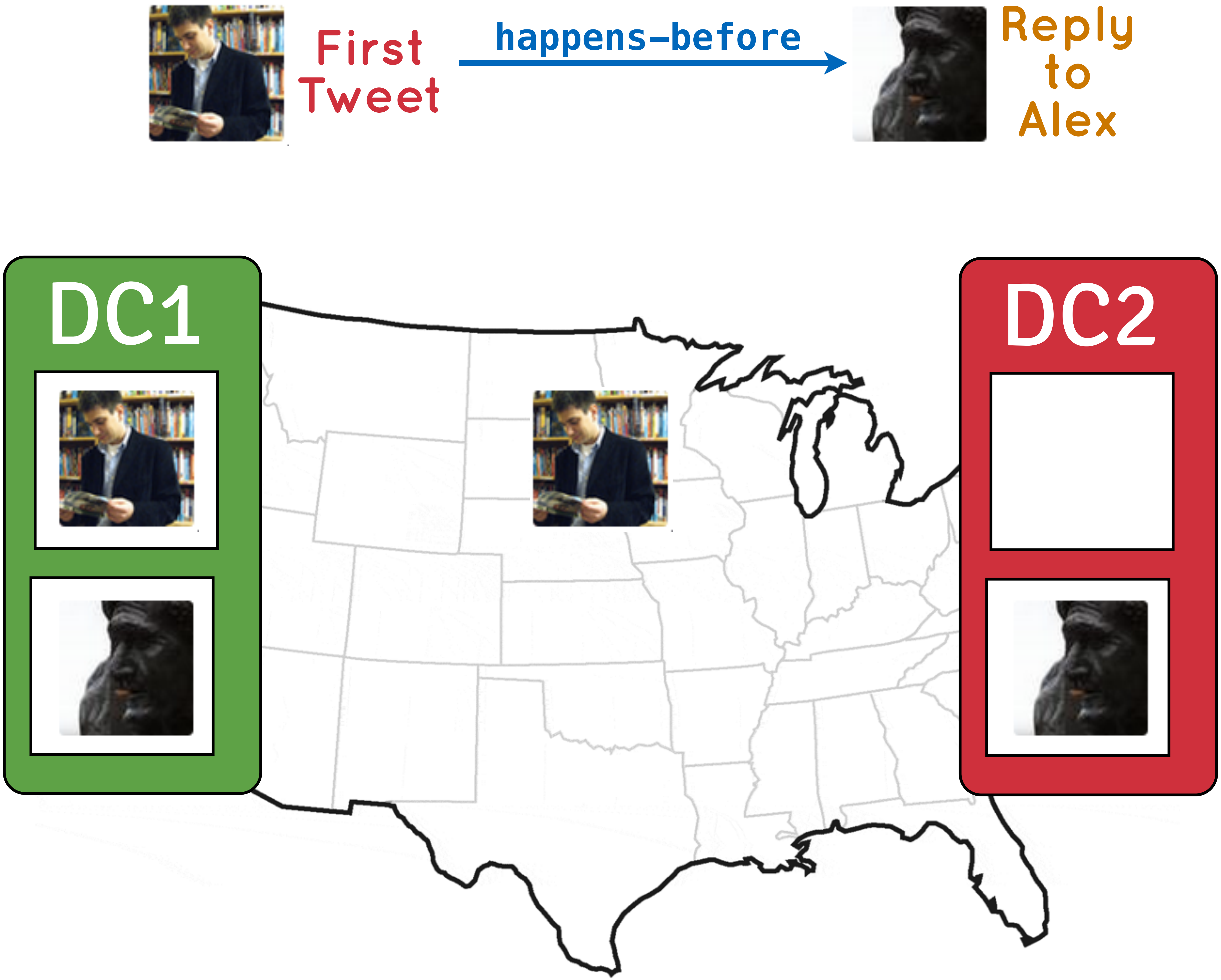

First Tweet
happens-before
Reply to Alex
DC1
DC2


First Tweet
happens-before
Reply to Alex
DC1
DC2

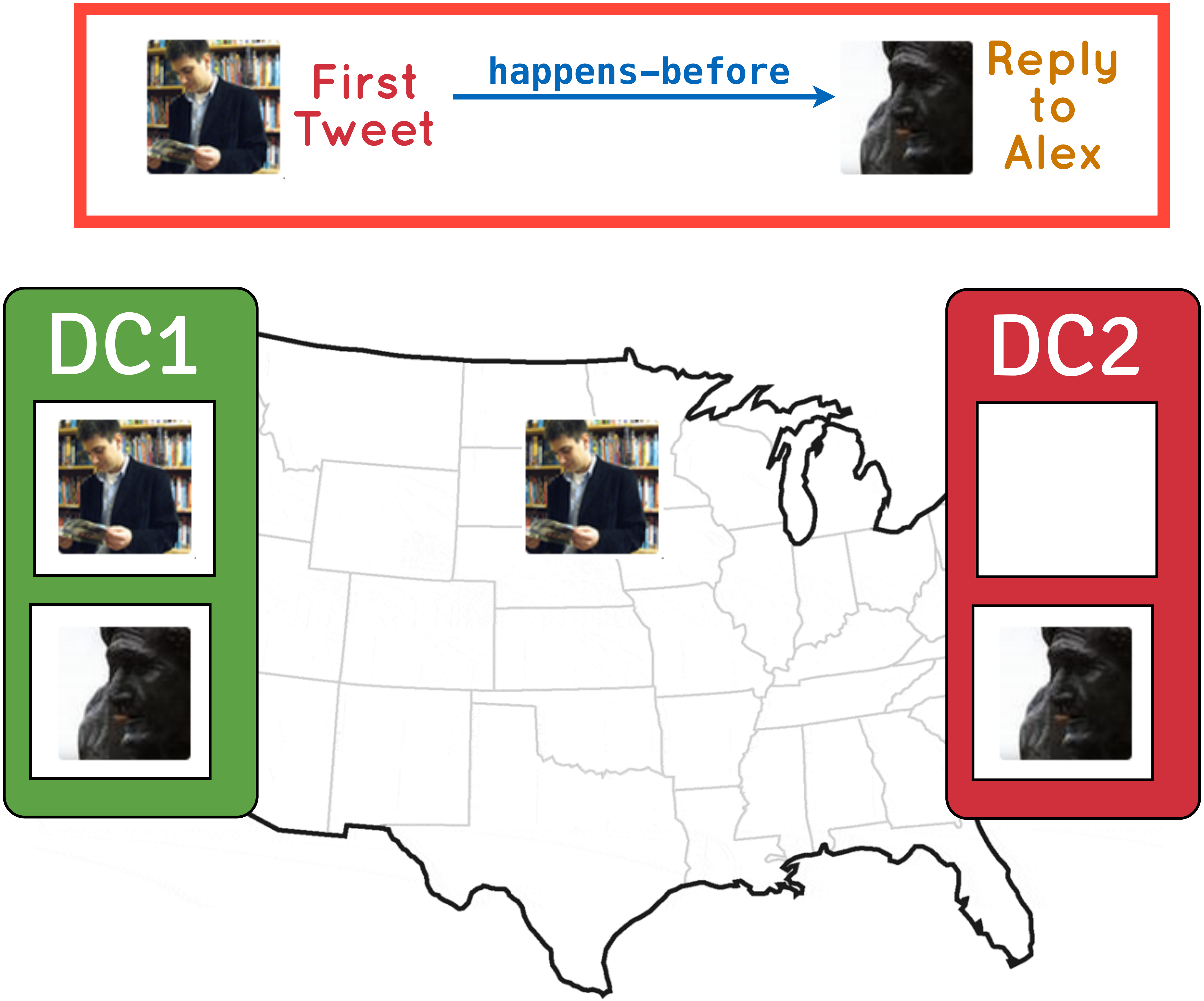

First Tweet
happens-before
Reply to Alex
DC1
DC2



First Tweet
happens-before
Reply to Alex
DC1
DC2



First Tweet
happens-before
Reply to Alex
DC1
DC2


First Tweet
happens-before
Reply to Alex
DC1
DC2

Two Tasks:

Two Tasks:

1.) Representing Order

How do we efficiently store causal ordering in the EC system?

Two Tasks:

1.) Representing Order

How do we efficiently store causal ordering in the EC system?

2.) Controlling Order

How do we control the visibility of new updates to the EC system?

Two Tasks:

1.) Representing Order

How do we efficiently store causal ordering in the EC system?

2.) Controlling Order

How do we control the visibility of new updates to the EC system?

Representing Order

Strawman: use vector clocks

[e.g., Bayou, Causal Memory]

Representing Order

Strawman: use vector clocks

[e.g., Bayou, Causal Memory]

First Tweet {  :1,  :0 }

Reply-to Alex {  :1,  :1 }

Representing Order

Strawman: use vector clocks

[e.g., Bayou, Causal Memory]

First Tweet {  :1,  :0 }

Reply-to Alex {  :1,  :1 }

Problem? Given missing dependency (from vector), what key should we check?

Representing Order

Strawman: use vector clocks

[e.g., Bayou, Causal Memory]

First Tweet {  :1,  :0 }

Reply-to Alex {  :1,  :1 }

Problem? Given missing dependency (from vector), what key should we check?

If I have <3,1>; where is <2,1>? <1,1>?

Write to same key?

Write to different key? Which?

Representing Order

Strawman: use dependency pointers

[e.g., Lazy Replication, COPS]

Representing Order

Strawman: use dependency pointers

[e.g., Lazy Replication, COPS]

First Tweet A @ timestamp 1092, dependencies = {}

Representing Order

Strawman: use dependency pointers

[e.g., Lazy Replication, COPS]

First Tweet	A @ timestamp 1092, dependencies = {}
Reply-to Alex	B @ timestamp 1109, dependencies=**{A@1092}**

Representing Order

Strawman: use dependency pointers

[e.g., Lazy Replication, COPS]

First Tweet	A @ timestamp 1092, dependencies = {}
Reply-to Alex	B @ timestamp 1109, dependencies=**{A@1092}**

Problem?

Representing Order

Strawman: use dependency pointers

[e.g., Lazy Replication, COPS]

First Tweet	A @ timestamp 1092, dependencies = {}
Reply-to Alex	B @ timestamp 1109, dependencies=**{A@1092}**

Problem?

A@1→B@2→C@3

Representing Order

Strawman: use dependency pointers

[e.g., Lazy Replication, COPS]

First Tweet	A @ timestamp 1092, dependencies = {}
Reply-to Alex	B @ timestamp 1109, dependencies=**{A@1092}**

Problem?

A@1→B@2→C@3

A@1	B@2	C@3

Representing Order

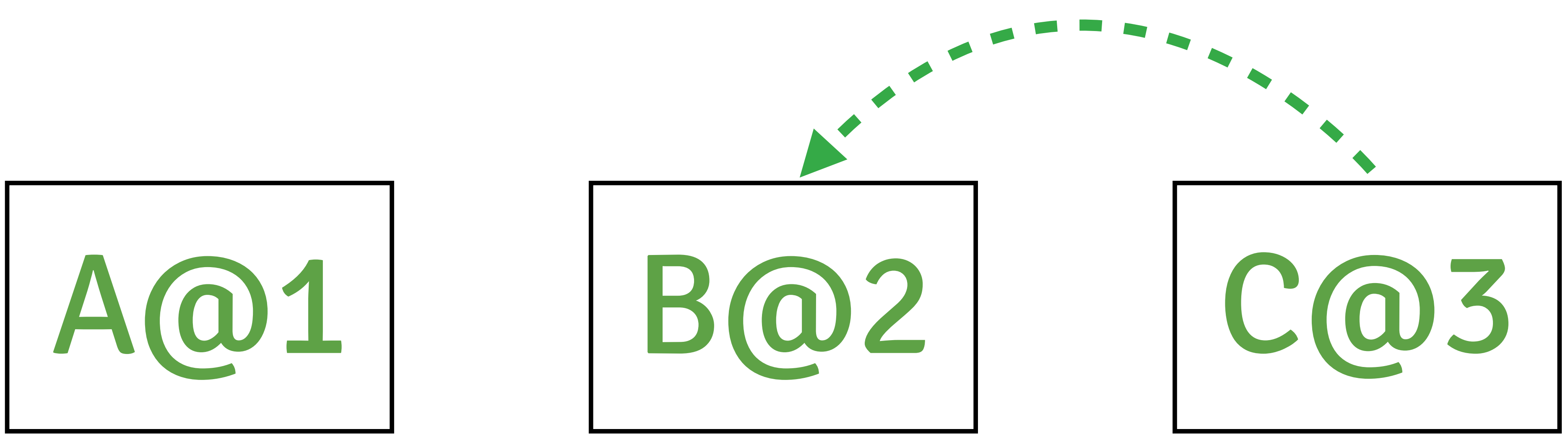
Strawman: use dependency pointers

[e.g., Lazy Replication, COPS]

First Tweet	A @ timestamp 1092, dependencies = {}
Reply-to Alex	B @ timestamp 1109, dependencies=**{A@1092}**

Problem?

A@1→B@2→C@3



Representing Order

Strawman: use dependency pointers

[e.g., Lazy Replication, COPS]

First Tweet

A @ timestamp 1092,
dependencies = {}

Reply-to Alex

B @ timestamp 1109,
dependencies=**{A@1092}**

Problem?

A@1→B@2→C@3

Representing Order

Strawman: use dependency pointers

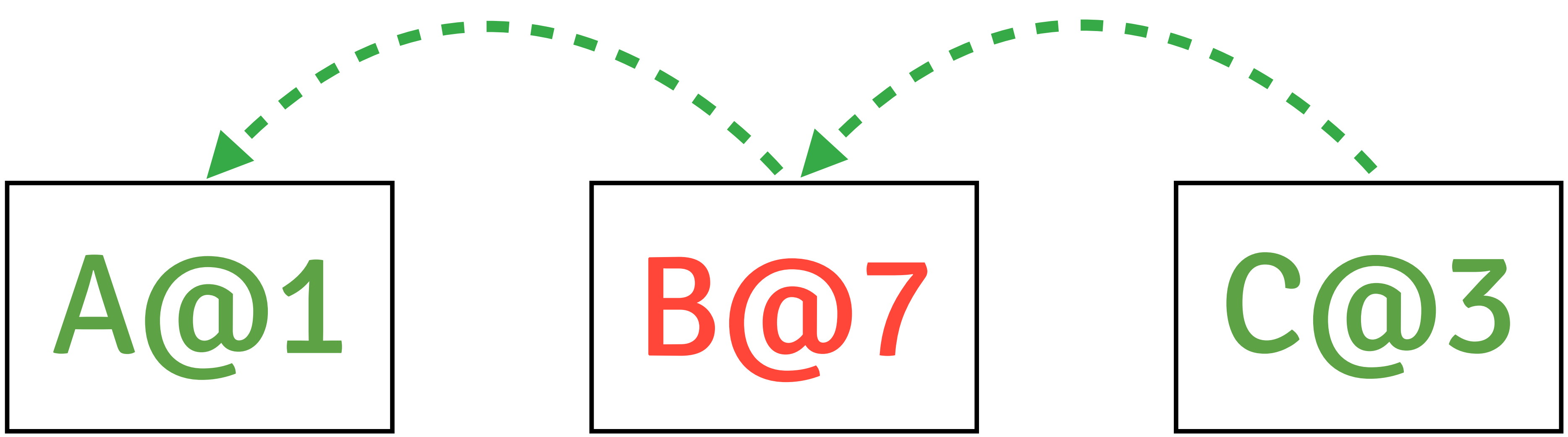
[e.g., Lazy Replication, COPS]

First Tweet — A @ timestamp 1092, dependencies = {}

Reply-to Alex — B @ timestamp 1109, dependencies=**{A@1092}**

Problem?

A@1→B@2→C@3



Representing Order

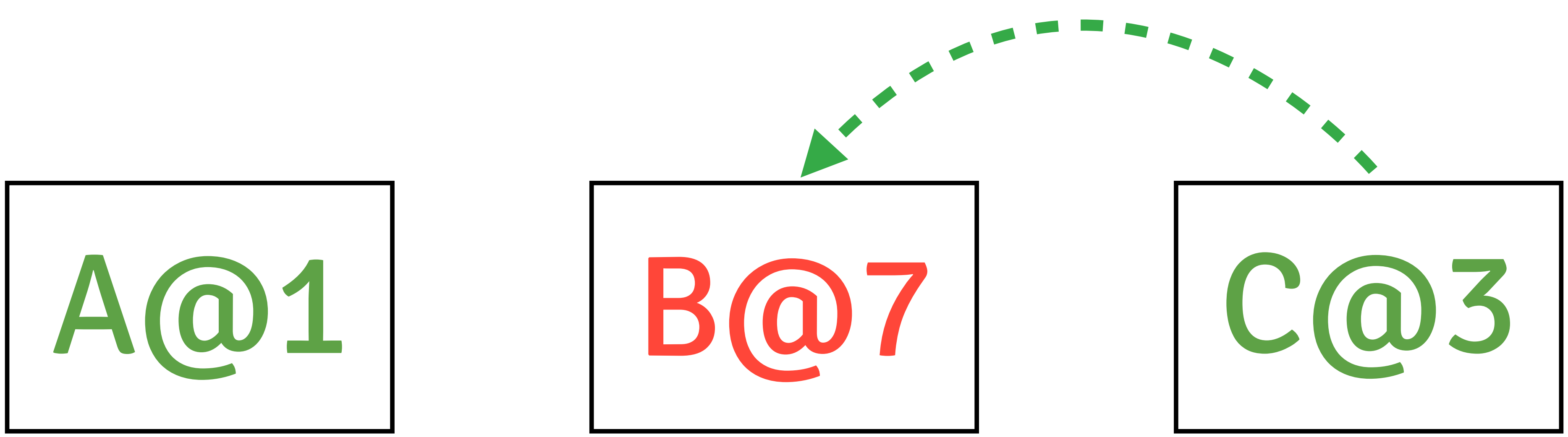
Strawman: use dependency pointers

[e.g., Lazy Replication, COPS]

First Tweet	A @ timestamp 1092, dependencies = {}
Reply-to Alex	B @ timestamp 1109, dependencies=**{A@1092}**

Problem?

A@1→B@2→C@3



Representing Order

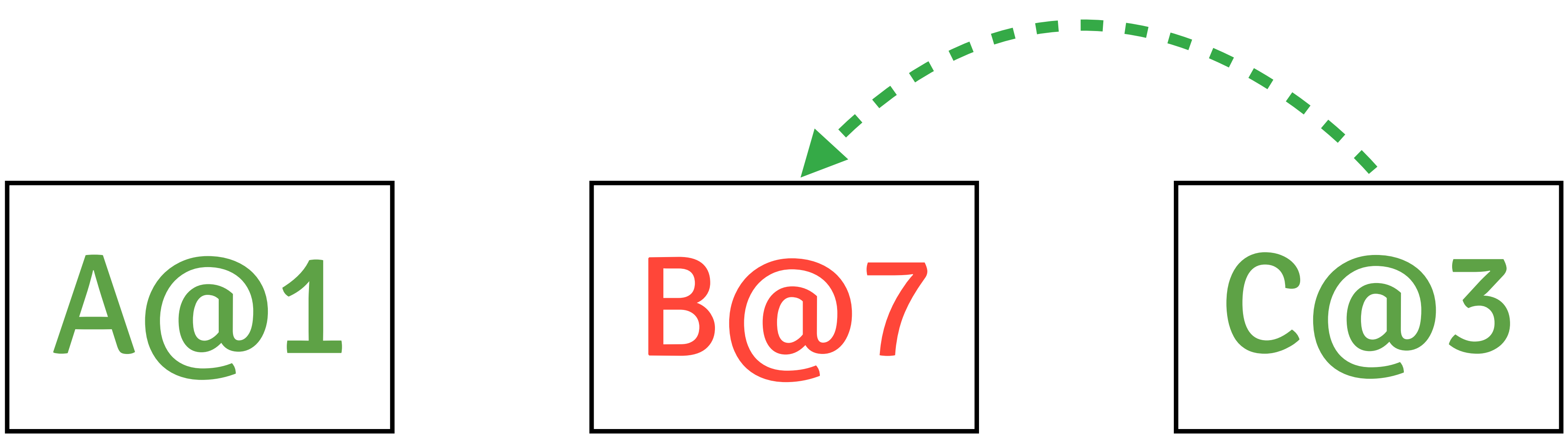
Strawman: use dependency pointers

[e.g., Lazy Replication, COPS]

First Tweet	A @ timestamp 1092, dependencies = {}
Reply-to Alex	B @ timestamp 1109, dependencies=**{A@1092}**

Problem? single pointers can be overwritten!

A@1→B@2→C@3



Representing Order

Representing Order

Strawman: use vector clocks

don't know what items to check

Representing Order

Strawman: use vector clocks

don't know what items to check

Strawman: use dependency pointers

single pointers can be overwritten

"overwritten histories"

Representing Order

Strawman: use vector clocks

don't know what items to check

Strawman: use dependency pointers

single pointers can be overwritten

"overwritten histories"

Strawman: use N^2 items for messaging

Representing Order

Strawman: use vector clocks

don't know what items to check

Strawman: use dependency pointers

single pointers can be overwritten

"overwritten histories"

Strawman: use N^2 items for messaging

highly inefficient!

Representing Order

Solution: store metadata about *causal cuts*

Representing Order

Solution: store metadata about *causal cuts*

DEFINITION 1. *A causal cut is a set of writes* C *such that* $\forall$ *writes* $w \in \bigcup_{c \in C} c.deps$, $\exists w' \in C$ *such that* $w'.key = w.key$ *and* $w' \not\rightarrow w$ *(equivalently, either* $w = w'$, $w \rightarrow w'$, *or* $w \parallel w$*).*

Representing Order

Solution: store metadata about *causal cuts*

DEFINITION 1. *A causal cut is a set of writes C such that $\forall$ writes $w \in \bigcup_{c \in C} c.deps$, $\exists w' \in C$ such that $w'.key = w.key$ and $w' \nrightarrow w$ (equivalently, either $w = w'$, $w \rightarrow w'$, or $w \parallel w$).*

short answer: consistent cut applied to data items; not quite the transitive closure

Representing Order

Solution: store metadata about *causal cuts*

short answer: consistent cut applied to data items; not quite the transitive closure

Representing Order

Solution: store metadata about *causal cuts*

short answer: consistent cut applied to data items; not quite the transitive closure

```
A@1→B@2→C@3
Causal cut for C@3: {B@2, A@1}
```

Representing Order

Solution: store metadata about *causal cuts*

short answer: consistent cut applied to data items; not quite the transitive closure

A@1 → B@2 → C@3

Causal cut for C@3: {B@2, A@1}

A@6 → B@17 → C@20

A@10 → B@12 ↗

Causal cut for C@20: {B@17, A@10}

Two Tasks:

1.) Representing Order

How do we efficiently store causal ordering in the EC system?

2.) Controlling Order

How do we control the visibility of new updates to the EC system?

Two Tasks:

1.) Representing Order

Shim stores causal cut summary along with every key due to overwrites and "unreliable" delivery

2.) Controlling Order

How do we control the visibility of new updates to the EC system?

Two Tasks:

1.) Representing Order

Shim stores causal cut summary along with every key due to overwrites and "unreliable" delivery

2.) Controlling Order

How do we control the visibility of new updates to the EC system?

Controlling Order

Controlling Order

Standard technique: reveal new writes to readers only when dependencies have been revealed

Inductively guarantee clients read from causal cut

Controlling Order

Standard technique: reveal new writes to readers only when dependencies have been revealed

Inductively guarantee clients read from causal cut

In bolt-on causal consistency, two challenges:

Controlling Order

Standard technique: reveal new writes to readers only when dependencies have been revealed

Inductively guarantee clients read from causal cut

In bolt-on causal consistency, two challenges:

Each shim has to check dependencies manually

Underlying store doesn't notify clients of new writes

Controlling Order

Standard technique: reveal new writes to readers only when dependencies have been revealed

Inductively guarantee clients read from causal cut

In bolt-on causal consistency, two challenges:

Each shim has to check dependencies manually

Underlying store doesn't notify clients of new writes

EC store may overwrite "stable" cut

Clients need to cache relevant cut to prevent overwrites

Controlling Order

Standard technique: reveal new writes to readers only when dependencies have been revealed

Inductively guarantee clients read from causal cut

In bolt-on causal consistency, two challenges:

Each shim has to check dependencies manually

Underlying store doesn't notify clients of new writes

EC store may overwrite "stable" cut

Clients need to cache relevant cut to prevent overwrites

Each shim has to check dependencies manually

EC store may overwrite “stable” cut

Each shim has to check dependencies manually

EC store may overwrite “stable” cut

Each shim has to check dependencies manually

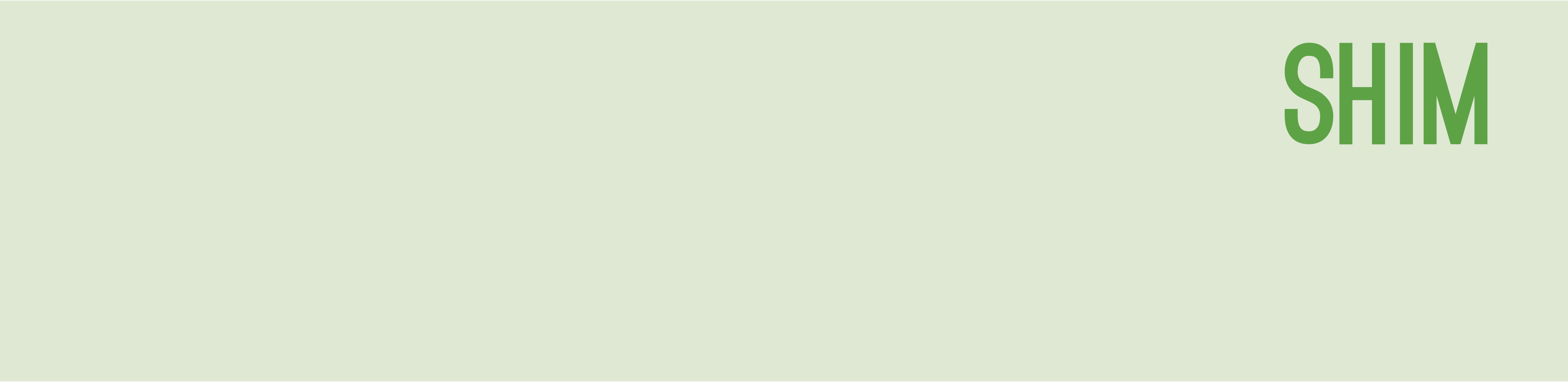
EC store may overwrite "stable" cut

Each shim has to check dependencies manually

EC store may overwrite “stable” cut

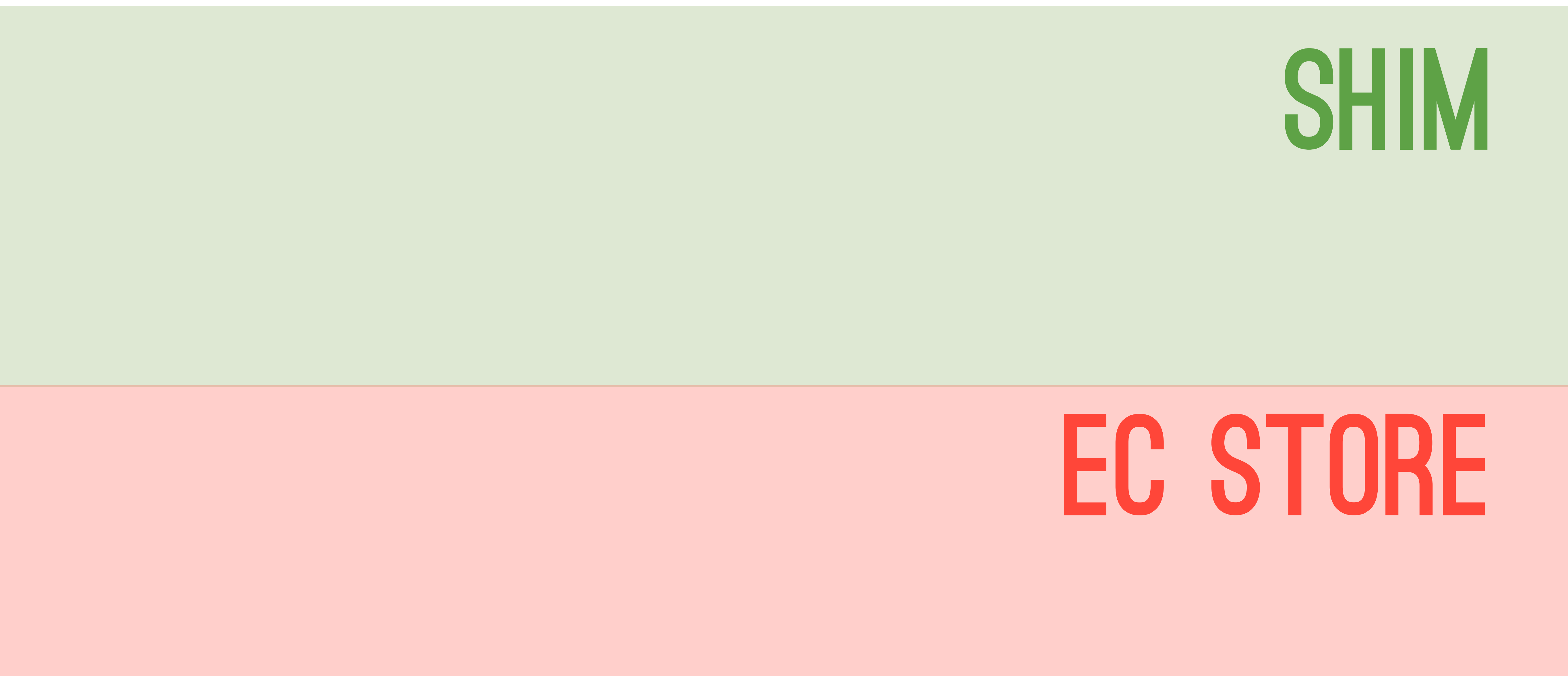
Each shim has to check dependencies manually

EC store may overwrite “stable” cut



Each shim has to check dependencies manually

EC store may overwrite "stable" cut



Each shim has to check dependencies manually

EC store may overwrite "stable" cut

read(B)

SHIM

EC STORE

Each shim has to check dependencies manually

EC store may overwrite "stable" cut

read(B)

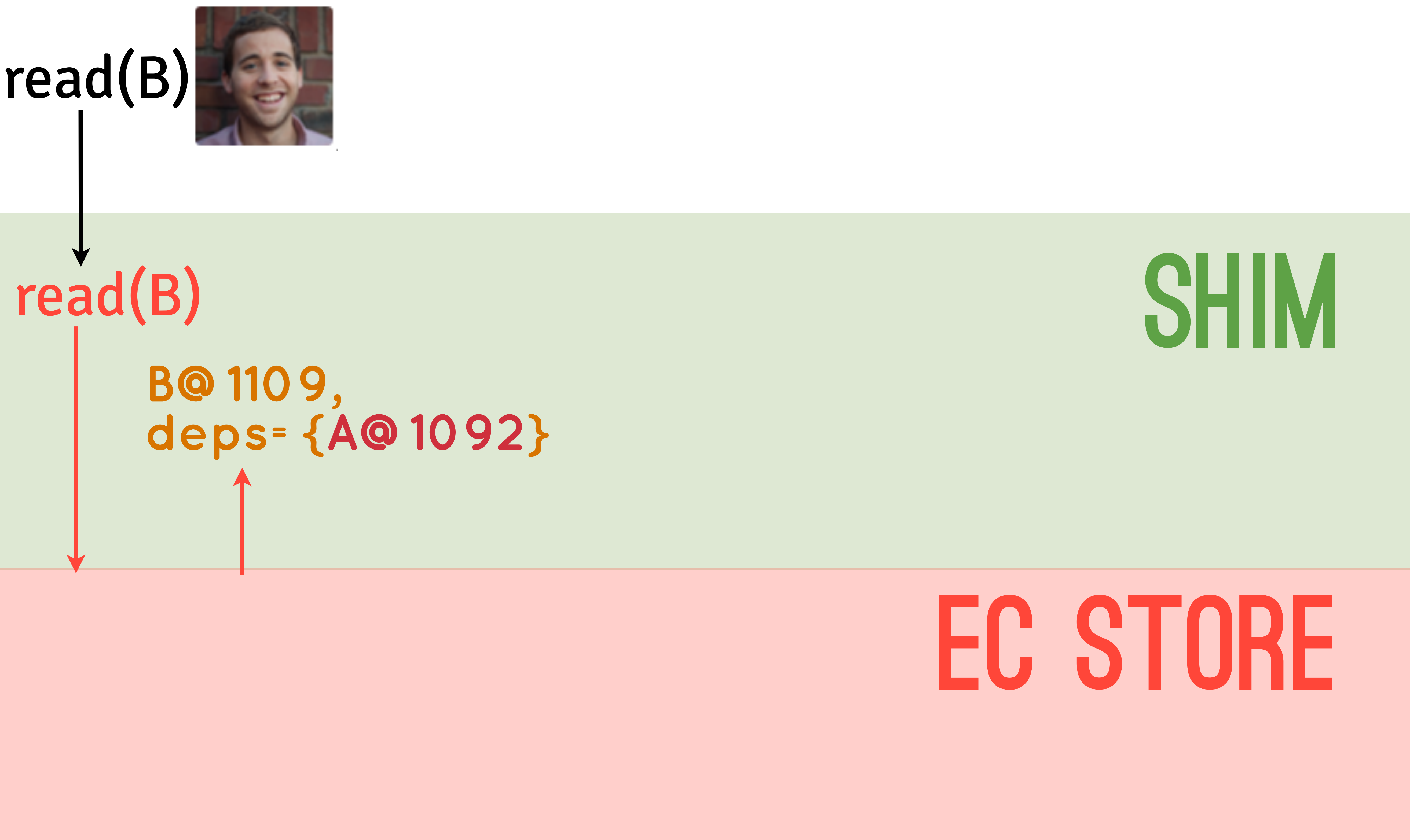
read(B)

SHIM

EC STORE

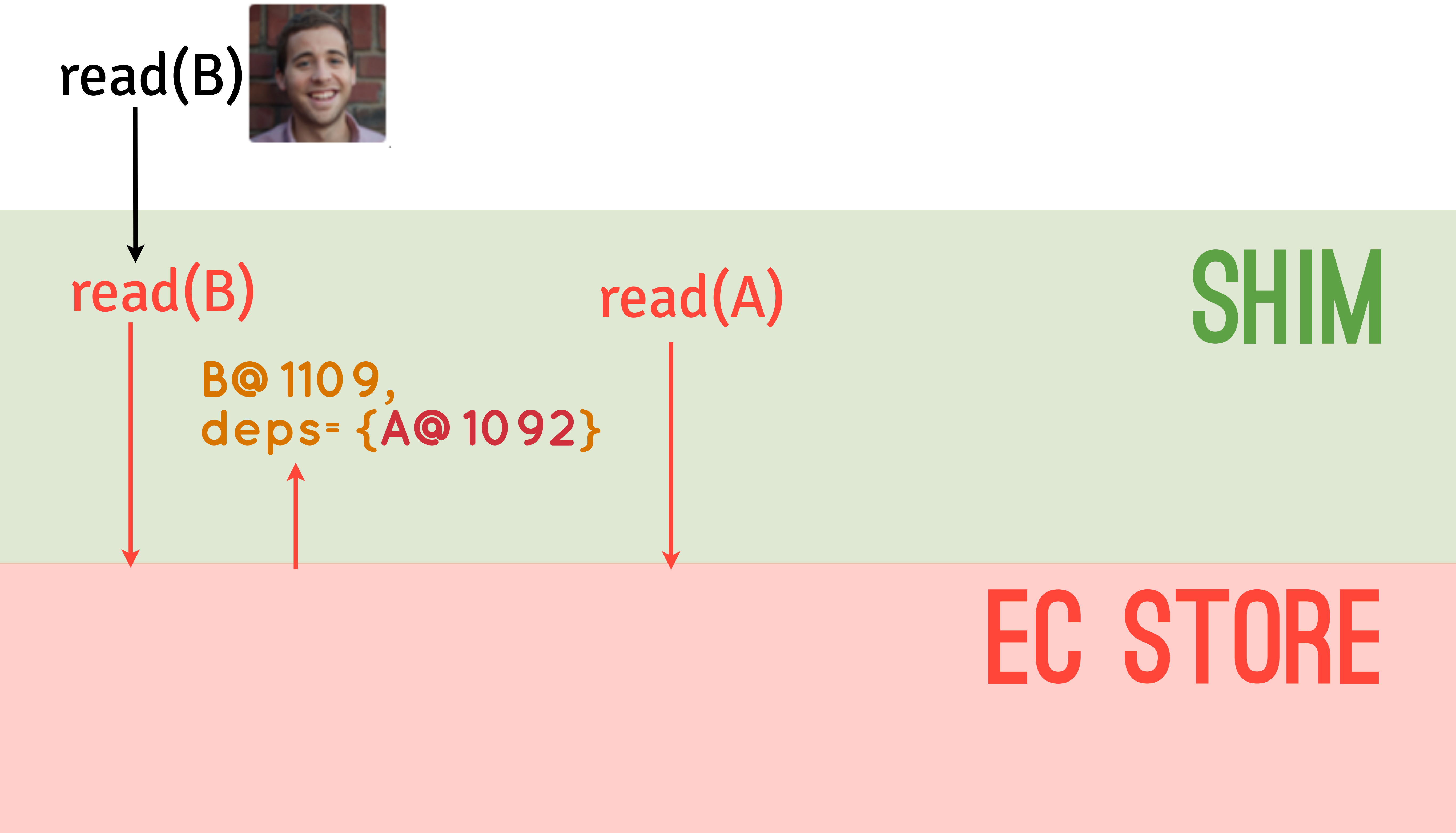
Each shim has to check dependencies manually

EC store may overwrite "stable" cut



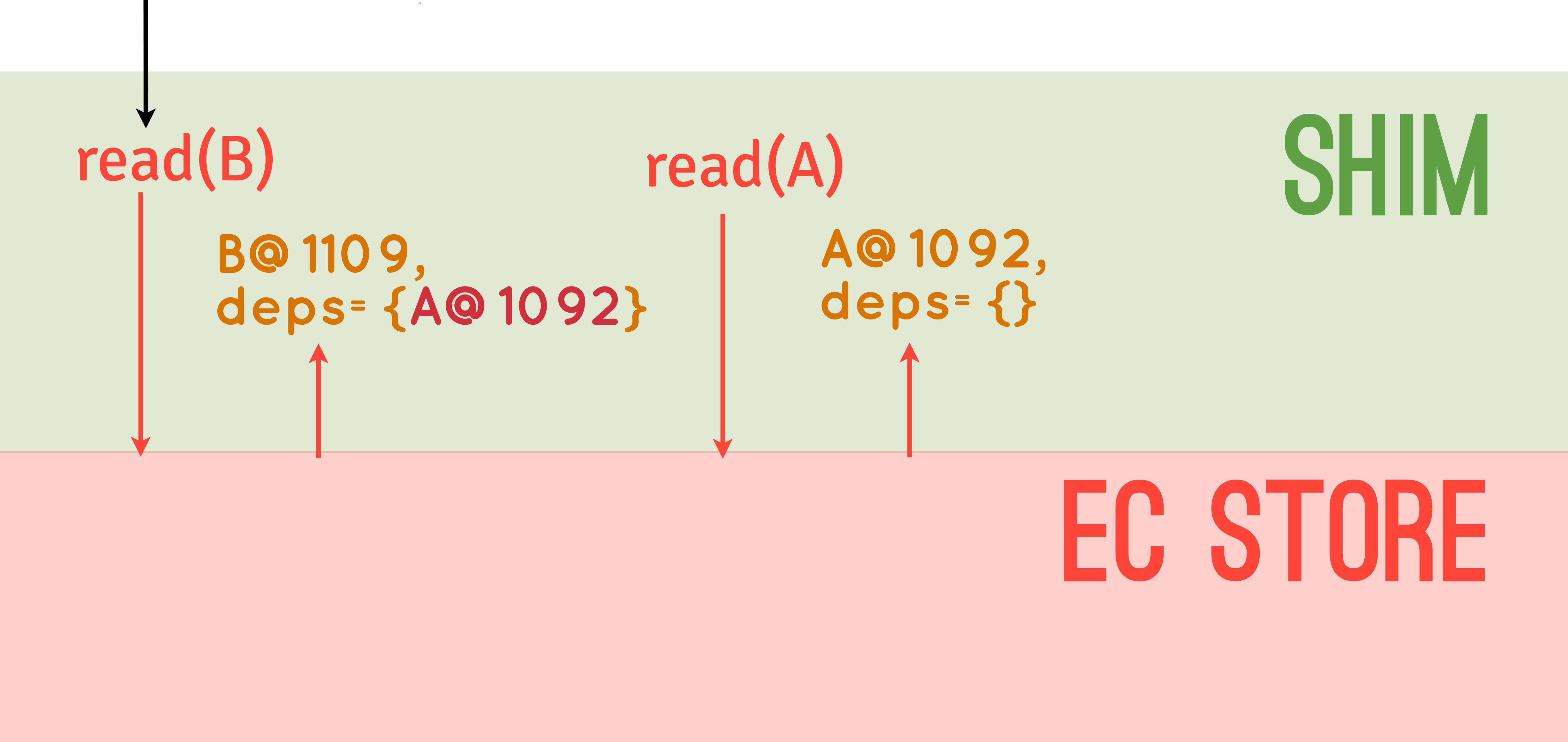
Each shim has to check dependencies manually

EC store may overwrite "stable" cut



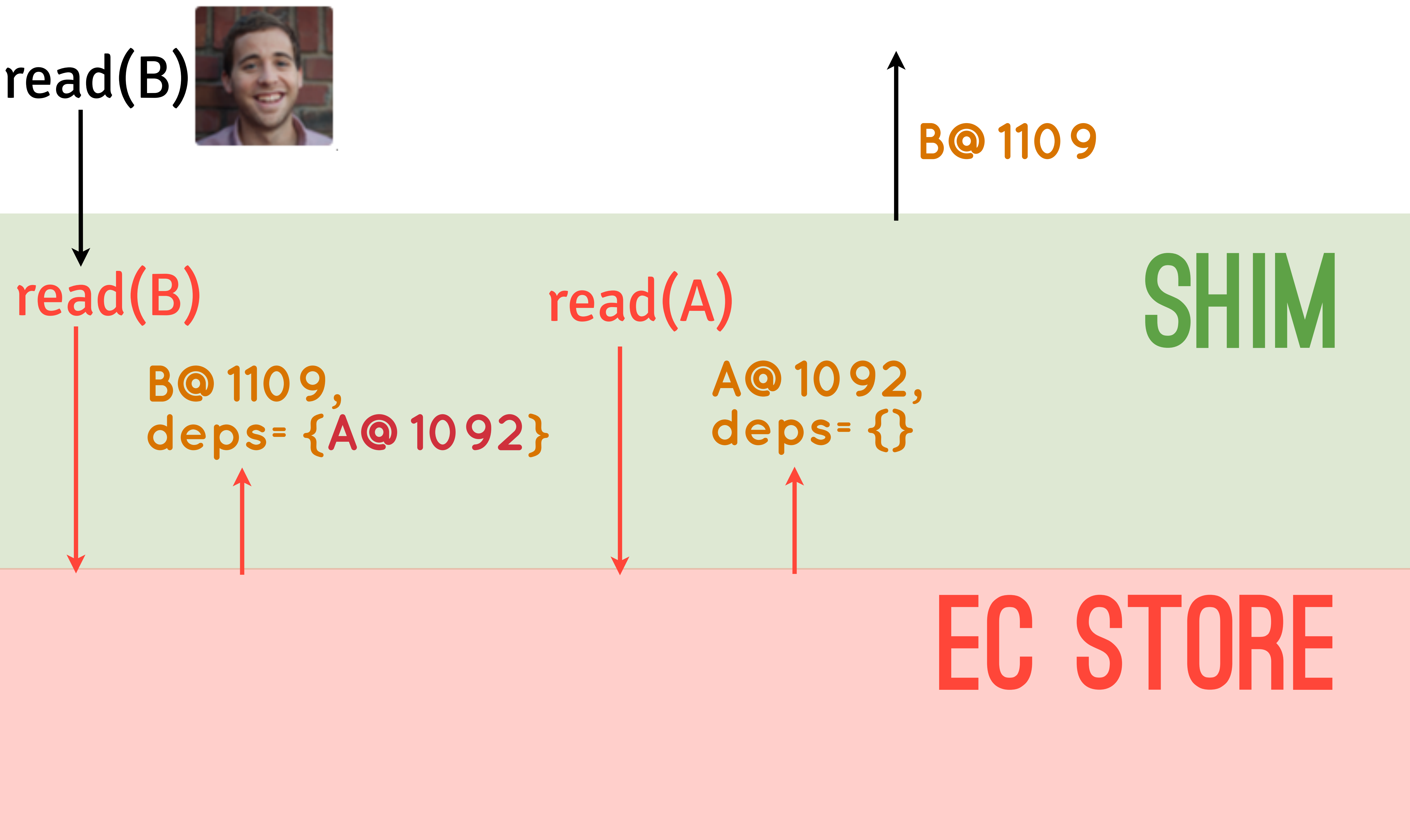
Each shim has to check dependencies manually

EC store may overwrite "stable" cut



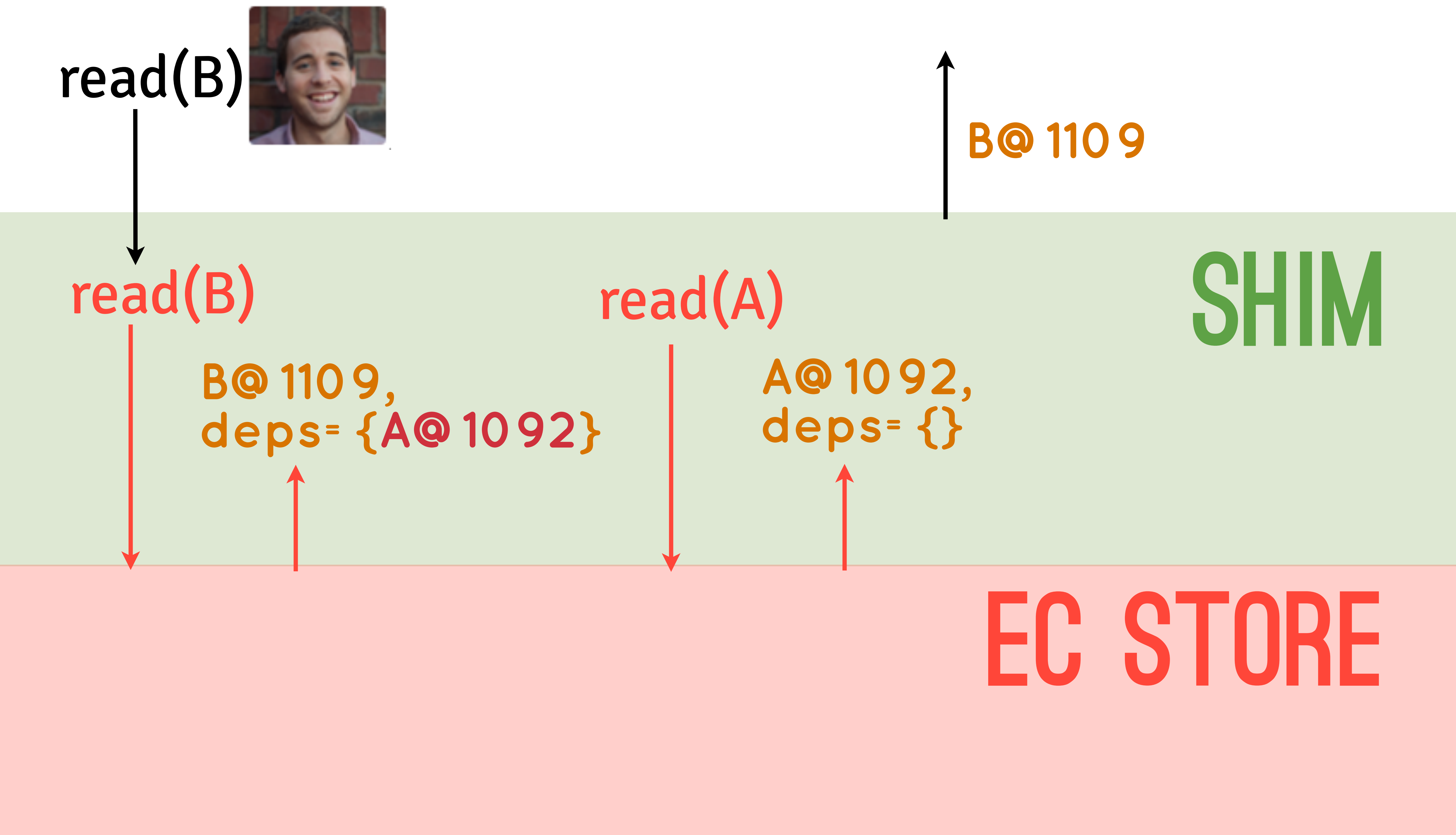
Each shim has to check dependencies manually

EC store may overwrite "stable" cut



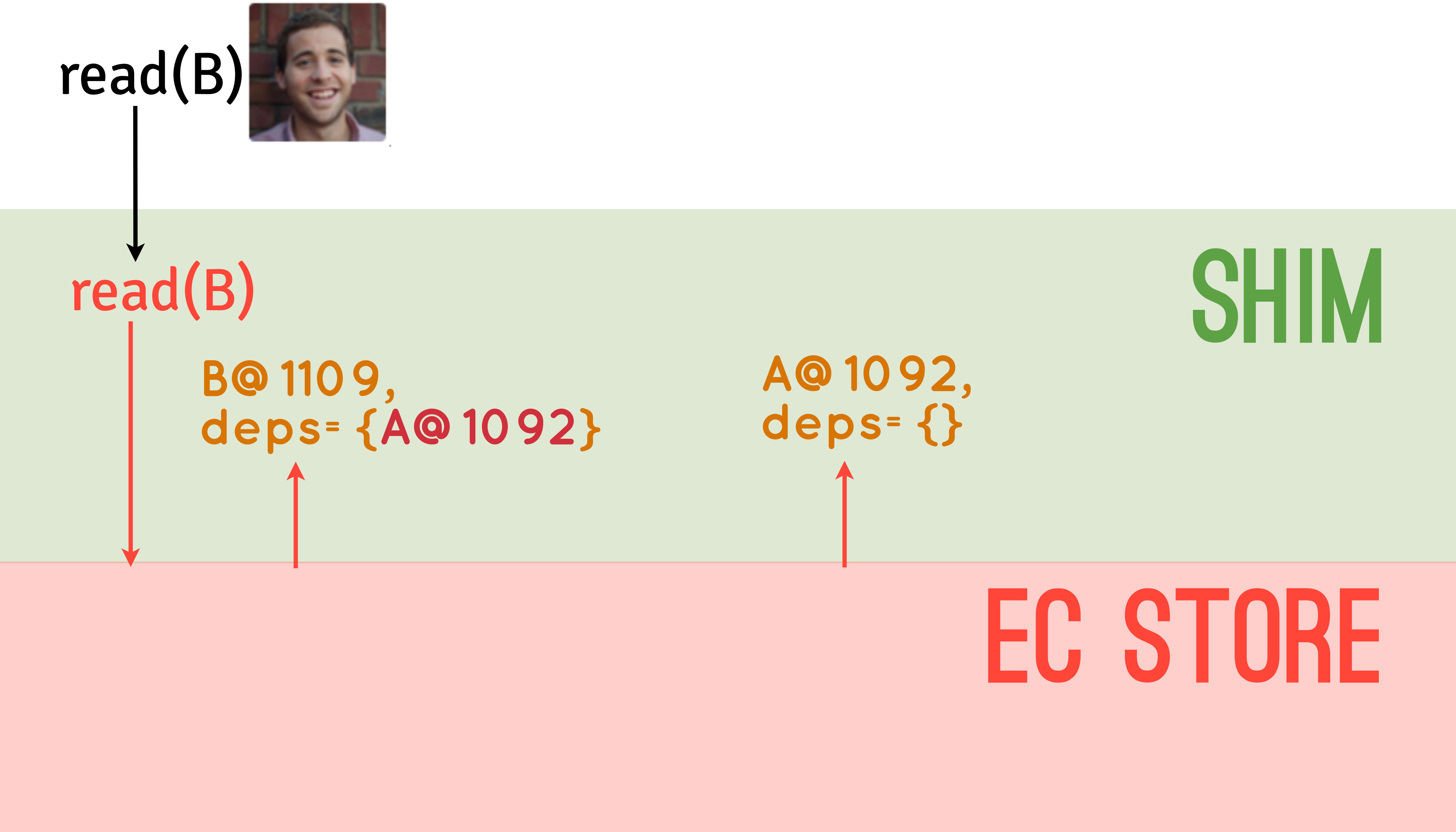
Each shim has to check dependencies manually

EC store may overwrite "stable" cut



Each shim has to check dependencies manually

EC store may overwrite “stable” cut



Each shim has to check dependencies manually

EC store may overwrite "stable" cut

read(B)

Cache this value for A!
EC store might overwrite it
with "unresolved" write

SHIM

read(B)

B@1109,
deps={A@1092}

A@1092,
deps={}

EC STORE

Two Tasks:

1.) Representing Order

Shim stores causal cut summary along with every key due to overwrites and "unreliable" delivery

2.) Controlling Order

How do we control the visibility of new updates to the EC system?

Two Tasks:

1.) Representing Order

Shim stores causal cut summary along with every key due to overwrites and "unreliable" delivery

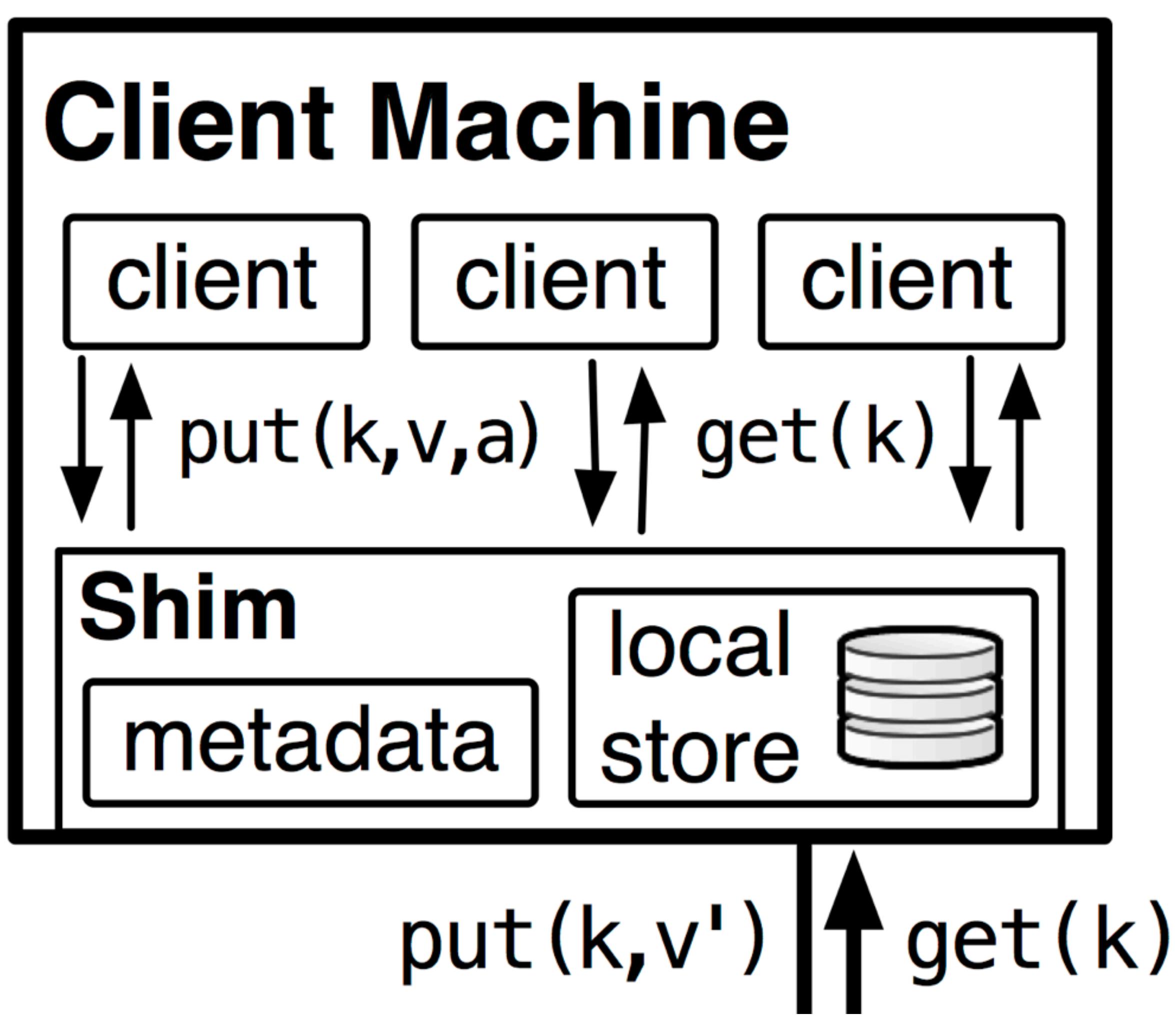
2.) Controlling Order

Shim performs dependency checks for client, caches dependencies

UPGRADED
CASSANDRA
TO
CAUSAL
CONSISTENCY

UPGRADED CASSANDRA TO CAUSAL CONSISTENCY

322 LINES JAVA FOR CORE SAFETY
CUSTOM SERIALIZATION
CLIENT-SIDE CACHING



MEDIAN

DATASET	CHAIN LENGTH
TWITTER	2
FLICKR	3
METAFILTER	6
TUAW	13

MEDIAN

DATASET	CHAIN LENGTH
TWITTER	2
FLICKR	3
METAFILTER	6
TUAW	13

MEDIAN

DATASET	CHAIN LENGTH
TWITTER	2
FLICKR	3
METAFILTER	6
TUAW	13

99TH PERCENTILE

TWITTER	40
FLICKR	44
METAFILTER	170
TUAW	62

MEDIAN

DATASET	CHAIN LENGTH
TWITTER	2
FLICKR	3
METAFILTER	6
TUAW	13

99TH PERCENTILE

TWITTER	40
FLICKR	44
METAFILTER	170
TUAW	62

MEDIAN

DATASET	CHAIN LENGTH	MESSAGE DEPTH
TWITTER	2	4
FLICKR	3	5
METAFILTER	6	18
TUAW	13	8

99TH PERCENTILE

TWITTER	40
FLICKR	44
METAFILTER	170
TUAW	62

MEDIAN

DATASET	CHAIN LENGTH	MESSAGE DEPTH
TWITTER	2	4
FLICKR	3	5
METAFILTER	6	18
TUAW	13	8

99TH PERCENTILE

TWITTER	40
FLICKR	44
METAFILTER	170
TUAW	62

MEDIAN

DATASET	CHAIN LENGTH	MESSAGE DEPTH
TWITTER	2	4
FLICKR	3	5
METAFILTER	6	18
TUAW	13	8

99TH PERCENTILE

TWITTER	40	230
FLICKR	44	100
METAFILTER	170	870
TUAW	62	100

MEDIAN

DATASET	CHAIN LENGTH	MESSAGE DEPTH
TWITTER	2	4
FLICKR	3	5
METAFILTER	6	18
TUAW	13	8

99TH PERCENTILE

TWITTER	40	230
FLICKR	44	100
METAFILTER	170	870
TUAW	62	100

MEDIAN

DATASET	CHAIN LENGTH	MESSAGE DEPTH	SERIALIZED SIZE (B)
TWITTER	2	4	169
FLICKR	3	5	201
METAFILTER	6	18	525
TUAW	13	8	275

99TH PERCENTILE

TWITTER	40	230
FLICKR	44	100
METAFILTER	170	870
TUAW	62	100

MEDIAN

DATASET	CHAIN LENGTH	MESSAGE DEPTH	SERIALIZED SIZE (B)
TWITTER	2	4	169
FLICKR	3	5	201
METAFILTER	6	18	525
TUAW	13	8	275

99TH PERCENTILE

TWITTER	40	230
FLICKR	44	100
METAFILTER	170	870
TUAW	62	100

MEDIAN

DATASET	CHAIN LENGTH	MESSAGE DEPTH	SERIALIZED SIZE (B)
TWITTER	2	4	169
FLICKR	3	5	201
METAFILTER	6	18	525
TUAW	13	8	275

99TH PERCENTILE

DATASET	CHAIN LENGTH	MESSAGE DEPTH	SERIALIZED SIZE (B)
TWITTER	40	230	5407
FLICKR	44	100	2447
METAFILTER	170	870	19375
TUAW	62	100	2438

MEDIAN

DATASET	CHAIN LENGTH	MESSAGE DEPTH	SERIALIZED SIZE (B)
TWITTER	2	4	169
FLICKR	3	5	201
METAFILTER	6	18	525
TUAW	13	8	275

99TH PERCENTILE

TWITTER	40	230	5407
FLICKR	44	100	2447
METAFILTER	170	870	19375
TUAW	62	100	2438

MEDIAN

DATASET	CHAIN LENGTH	MESSAGE DEPTH	SERIALIZED SIZE (B)
TWITTER	2	4	169
FLICKR	3	5	201
METAFILTER	6	16	525
TUAW	13	8	275

Most chains are small

Metadata often < 1KB

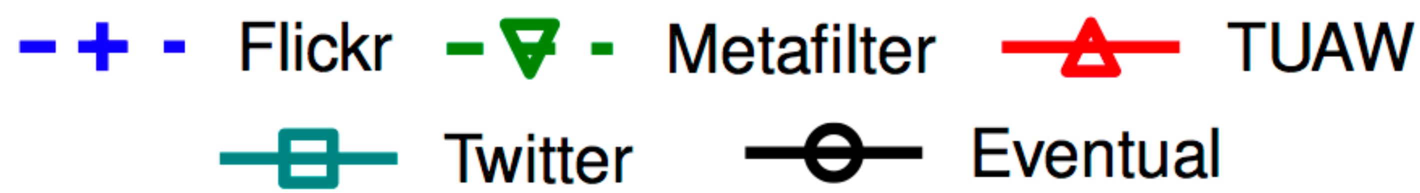
Power laws mean some chains are difficult

99TH PERCENTILE

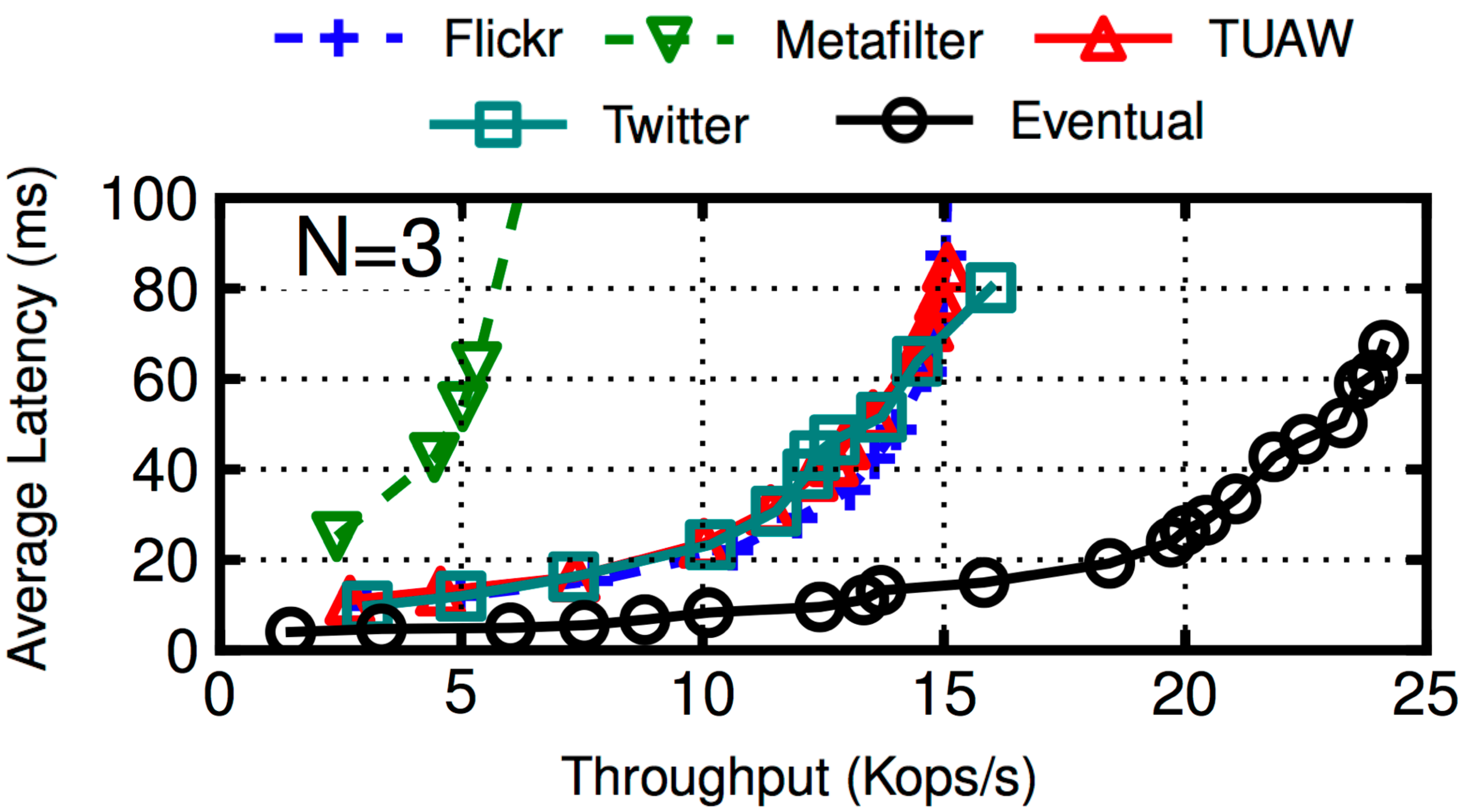
TWITTER	40	230	5407
FLICKR	44	100	2447
METAFILTER	170	870	19375
TUAW	62	100	2438

Strategy 1: Resolve dependencies at read time

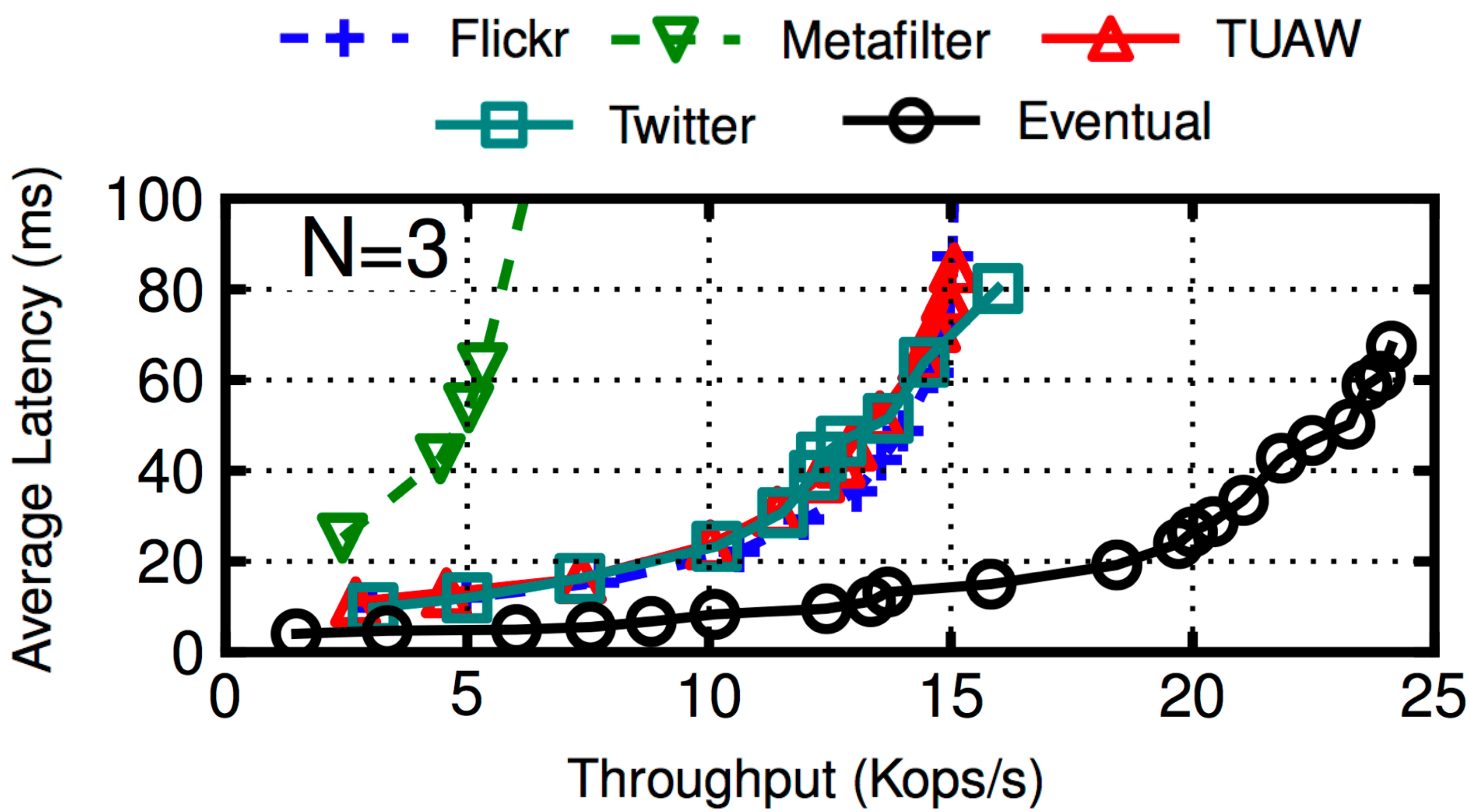
Strategy 1: Resolve dependencies at read time



Strategy 1: Resolve dependencies at read time



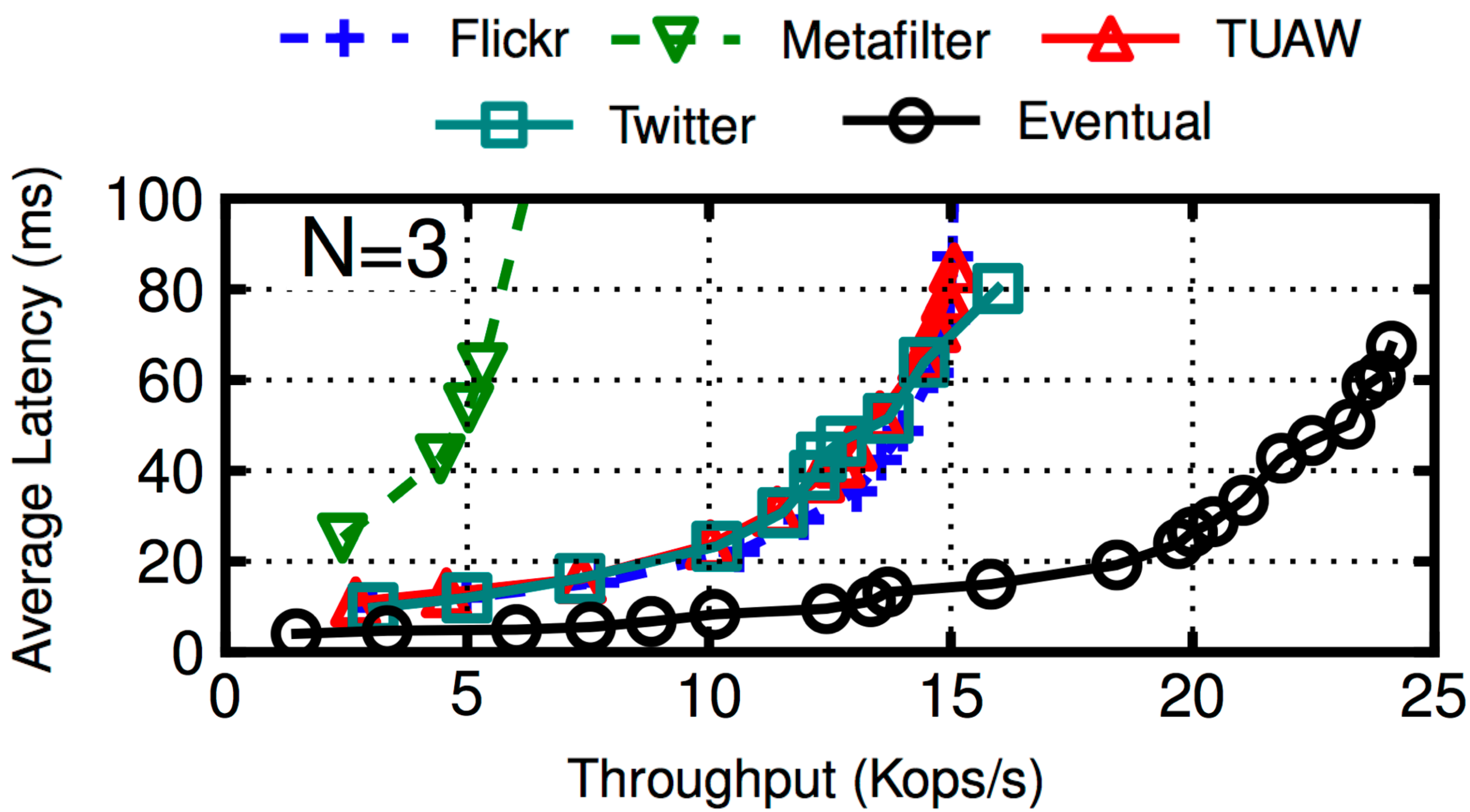
Strategy 1: Resolve dependencies at read time



Often (but not always) within 40% of eventual

Long chains hurt throughput

Strategy 1: Resolve dependencies at read time



Often (but not always) within 40% of eventual
Long chains hurt throughput

N.B. Locality in YCSB workload greatly helps read performance; dependencies (or replacements) often cached
(used 100x default # keys, but still likely to have concurrent write in cache)

A thought…

Causal consistency trades **visibility** for **safety**

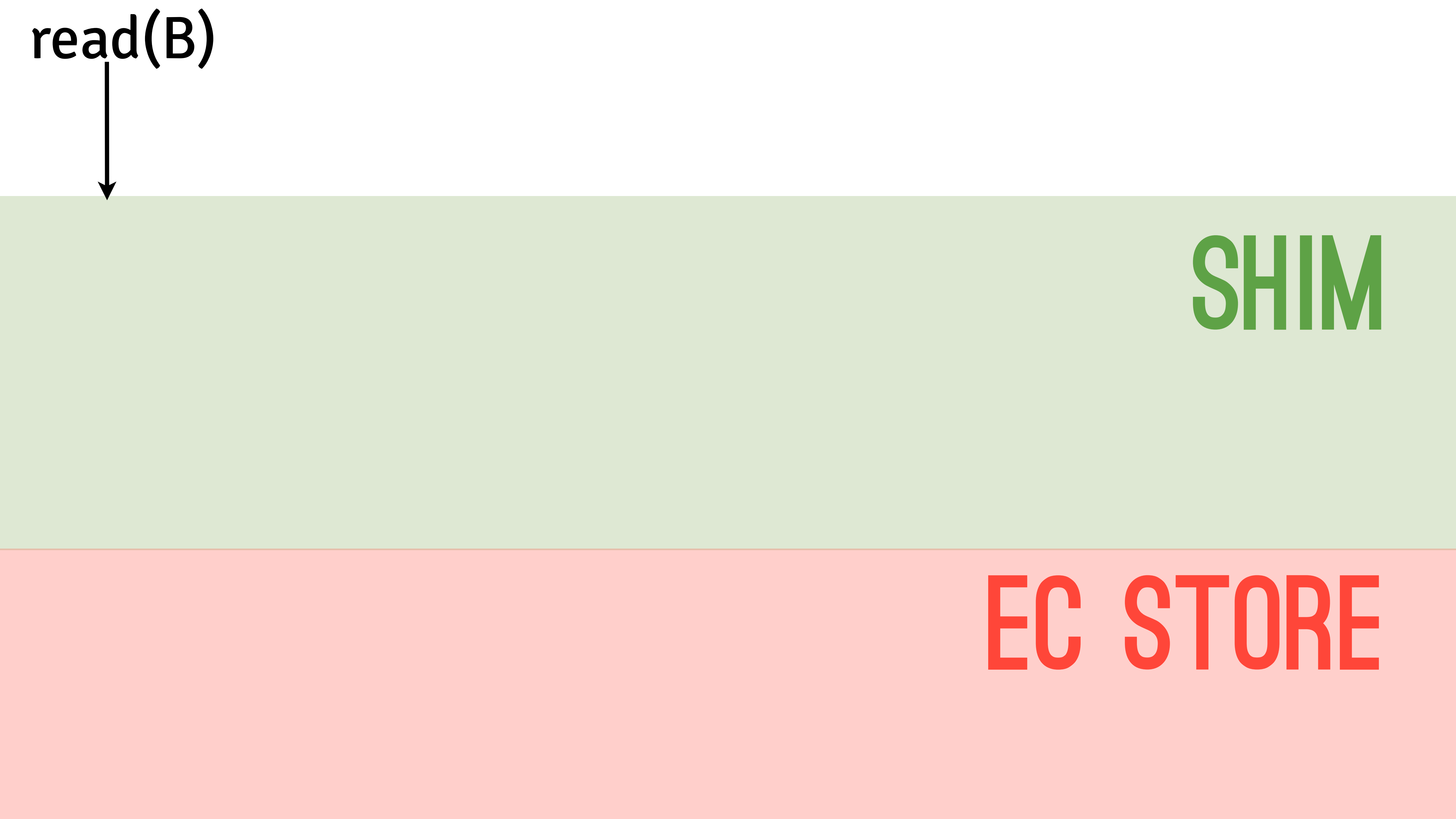
How far can we push this visibility?

What if we serve entirely from cache and fetch new data *asynchronously*?

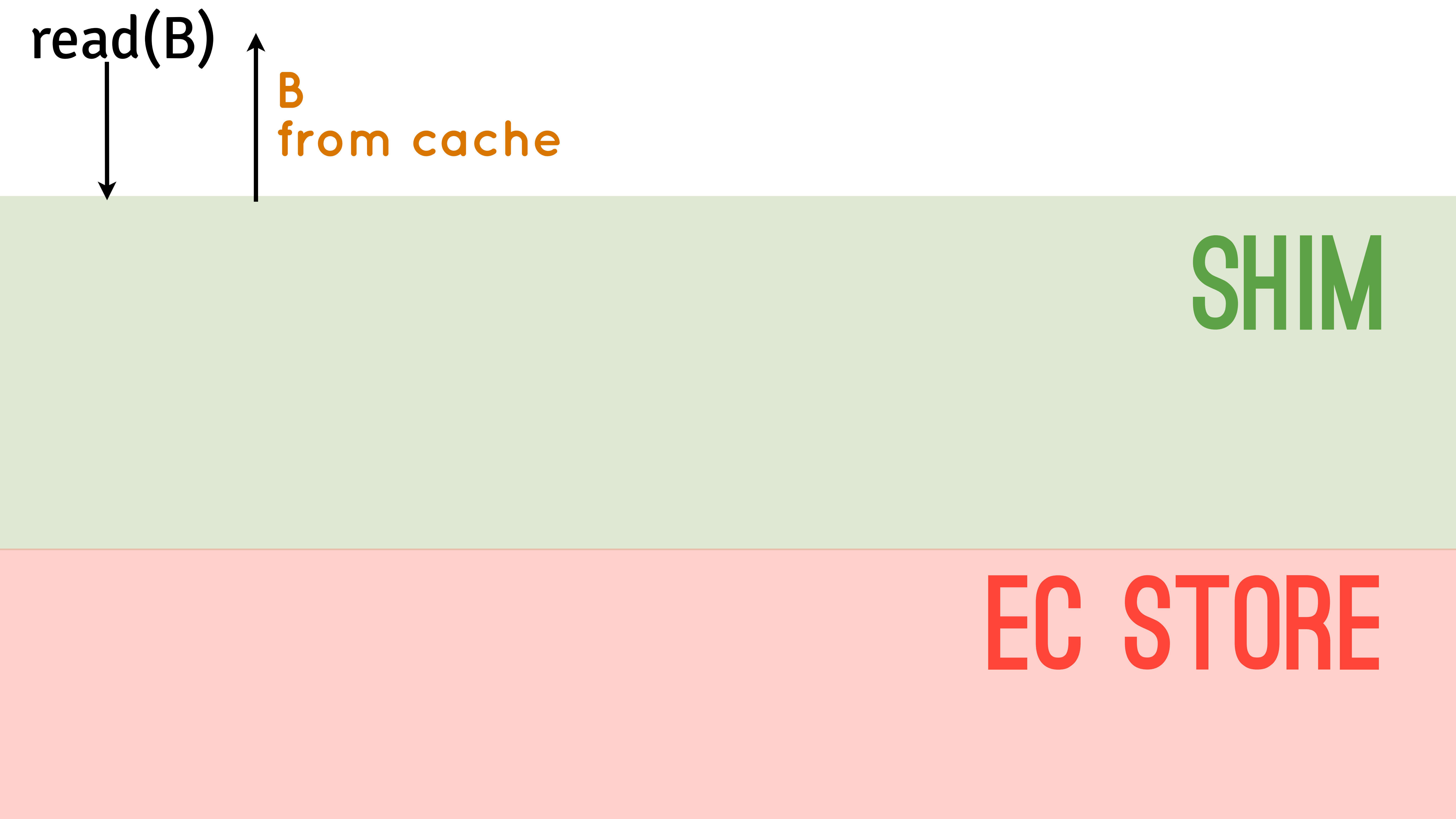
SHIM

EC STORE

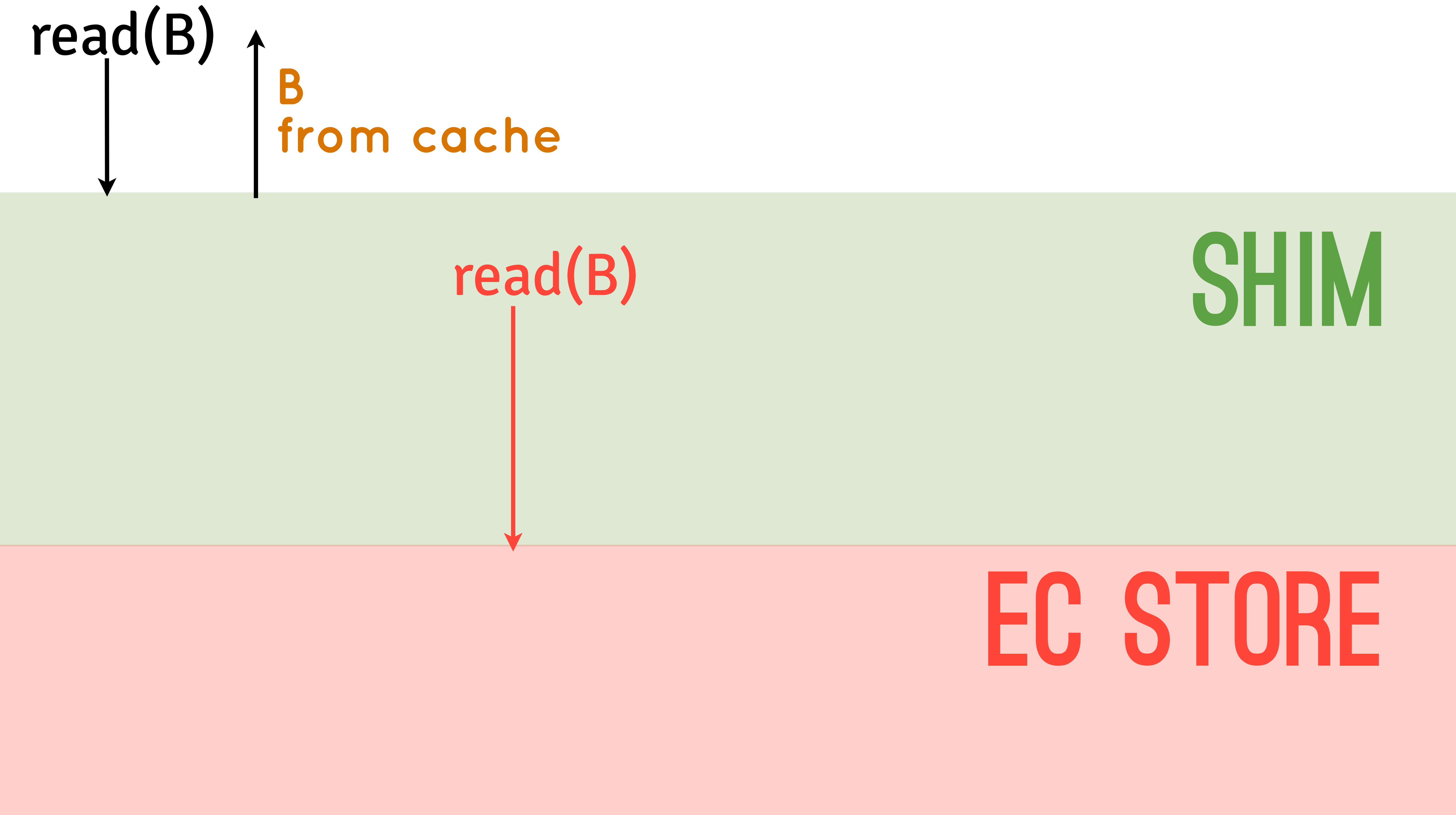
What if we serve entirely from cache and fetch new data *asynchronously*?



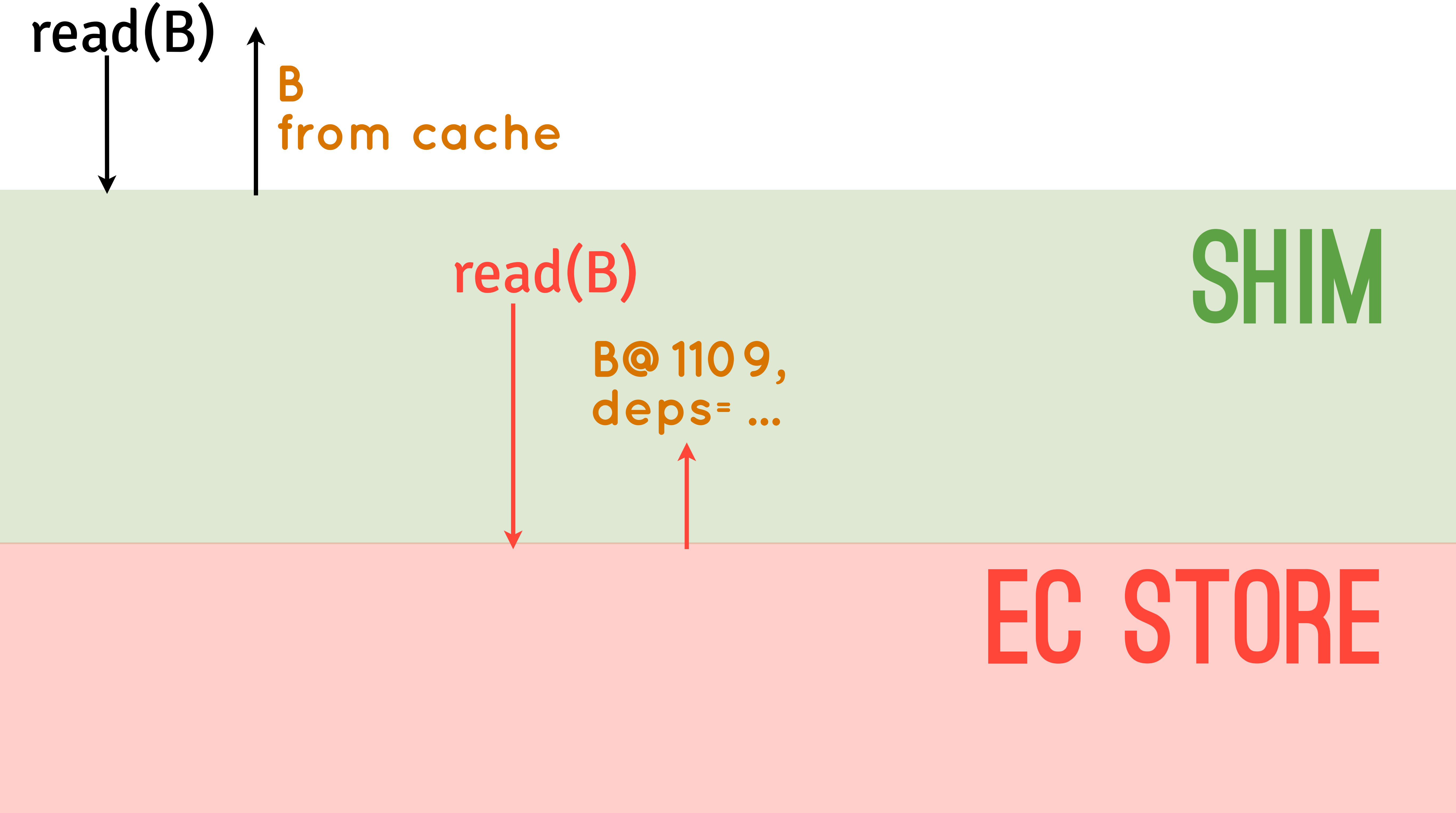
What if we serve entirely from cache and fetch new data *asynchronously*?



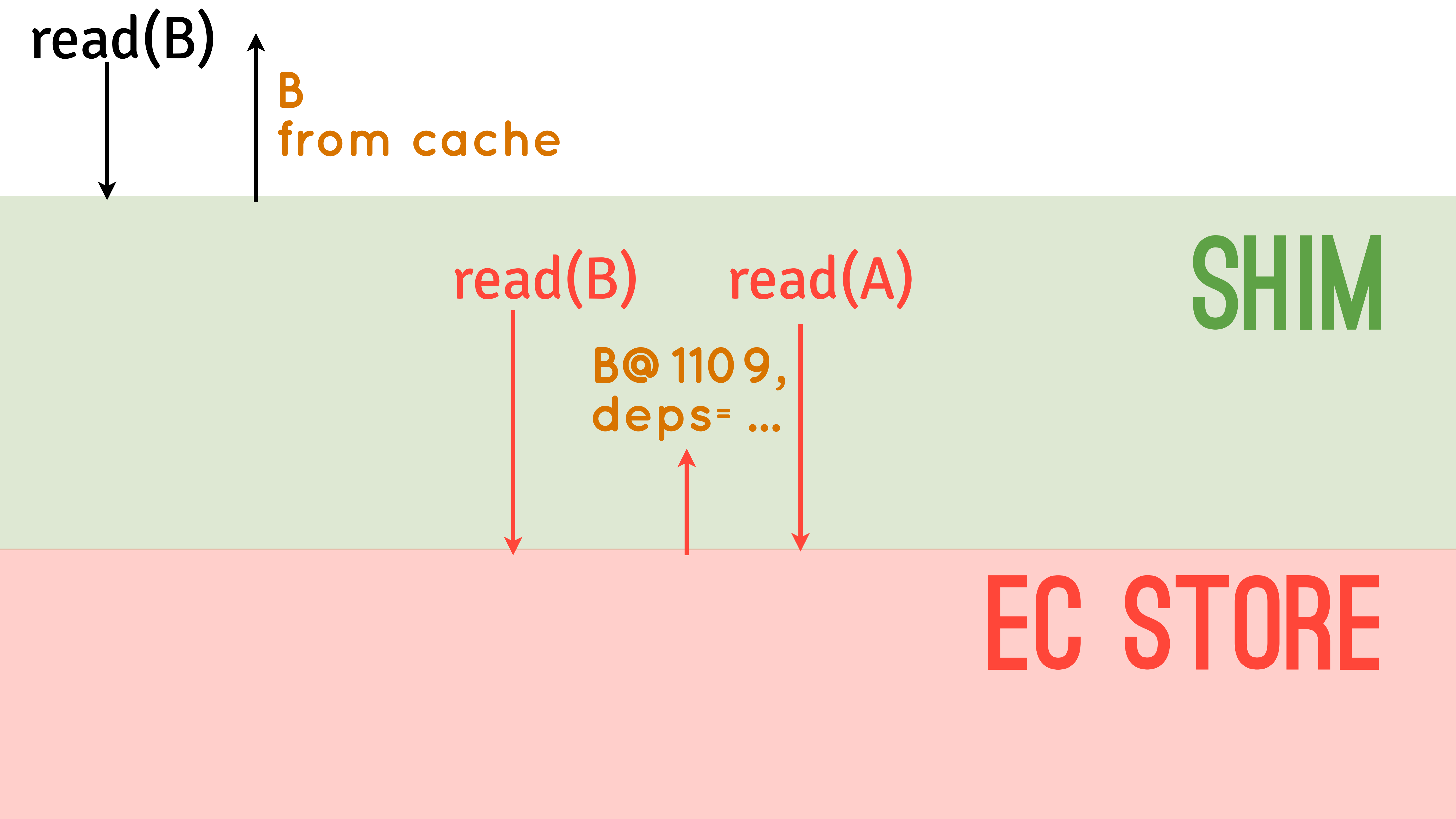
What if we serve entirely from cache and fetch new data *asynchronously*?



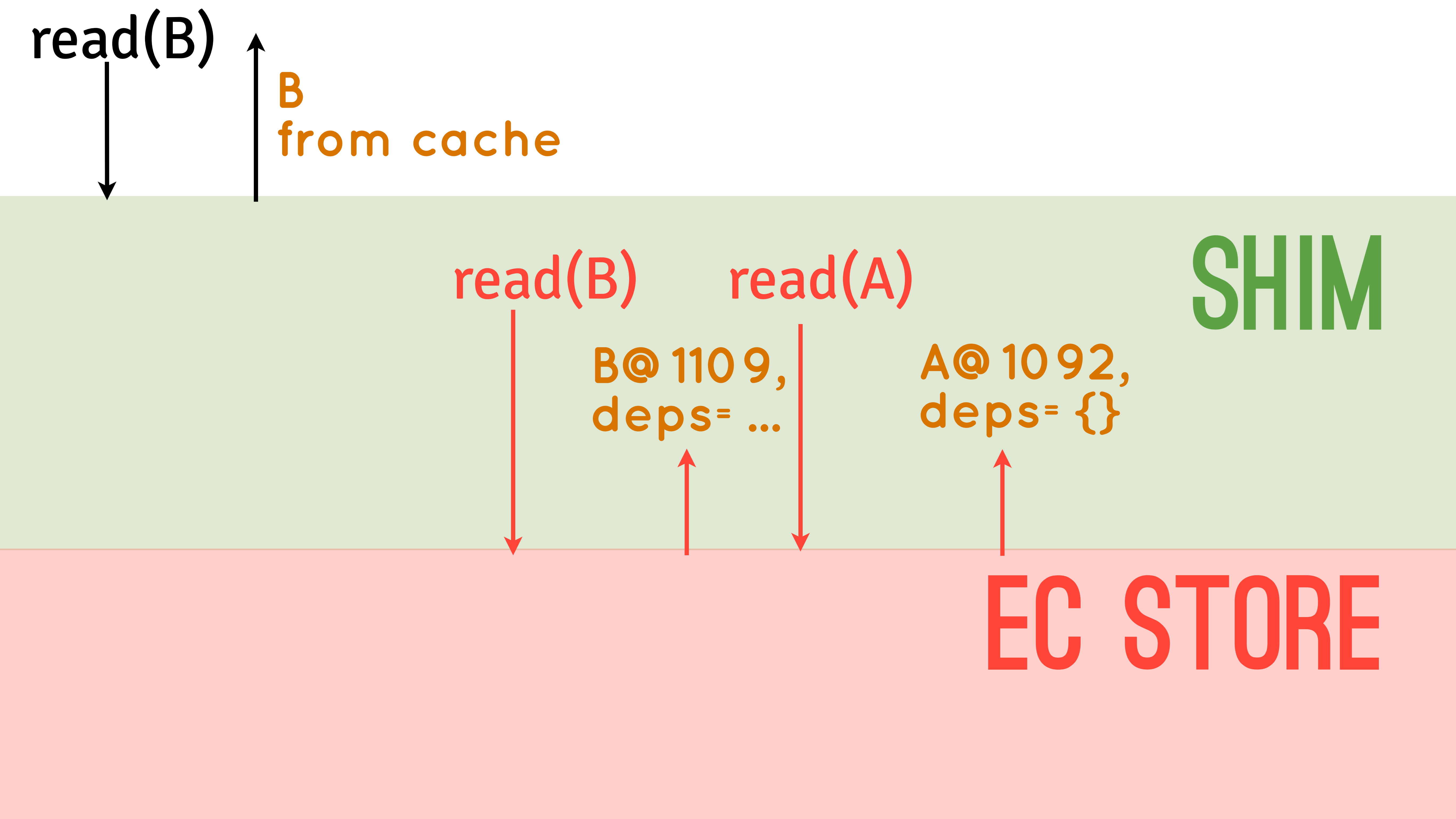
What if we serve entirely from cache and fetch new data *asynchronously*?



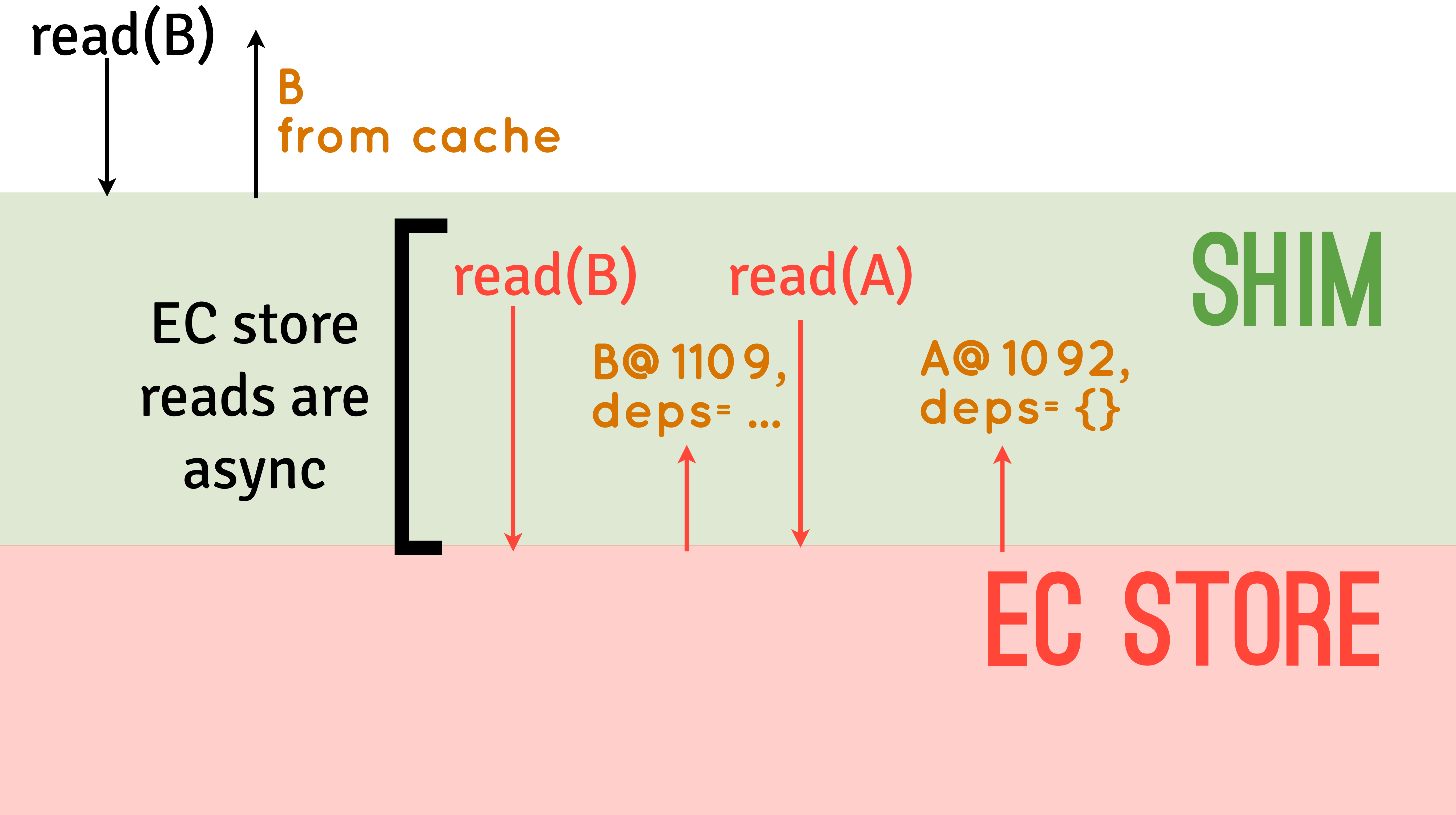
What if we serve entirely from cache and fetch new data *asynchronously*?



What if we serve entirely from cache and fetch new data *asynchronously*?



What if we serve entirely from cache and fetch new data *asynchronously*?



A thought...

Causal consistency trades **visibility** for **safety**

How far can we push this visibility?

What if we serve reads entirely from cache and fetch new data *asynchronously*?

A thought...

Causal consistency trades **visibility** for **safety**

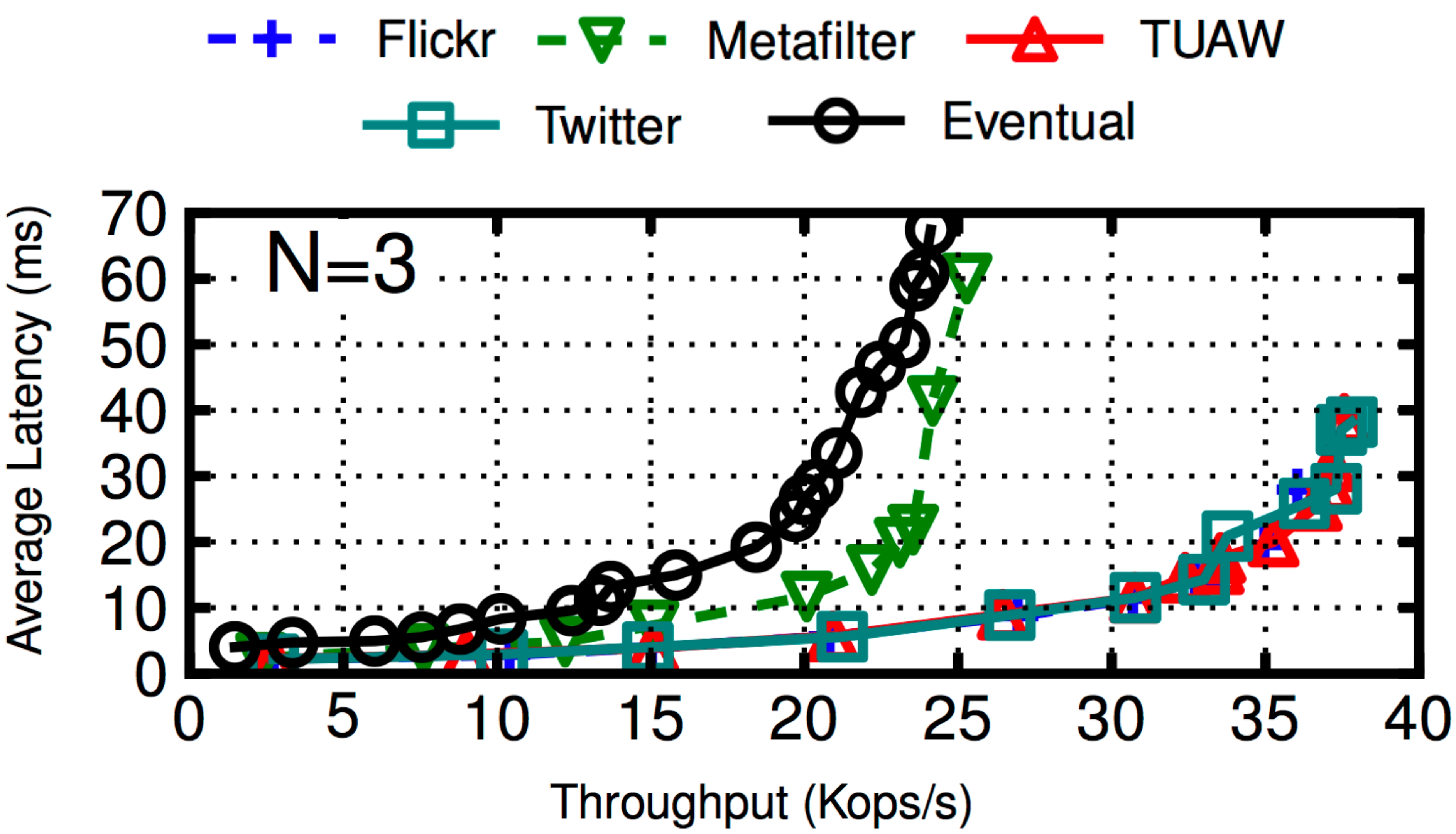
How far can we push this visibility?

What if we serve reads entirely from cache and fetch new data *asynchronously*?

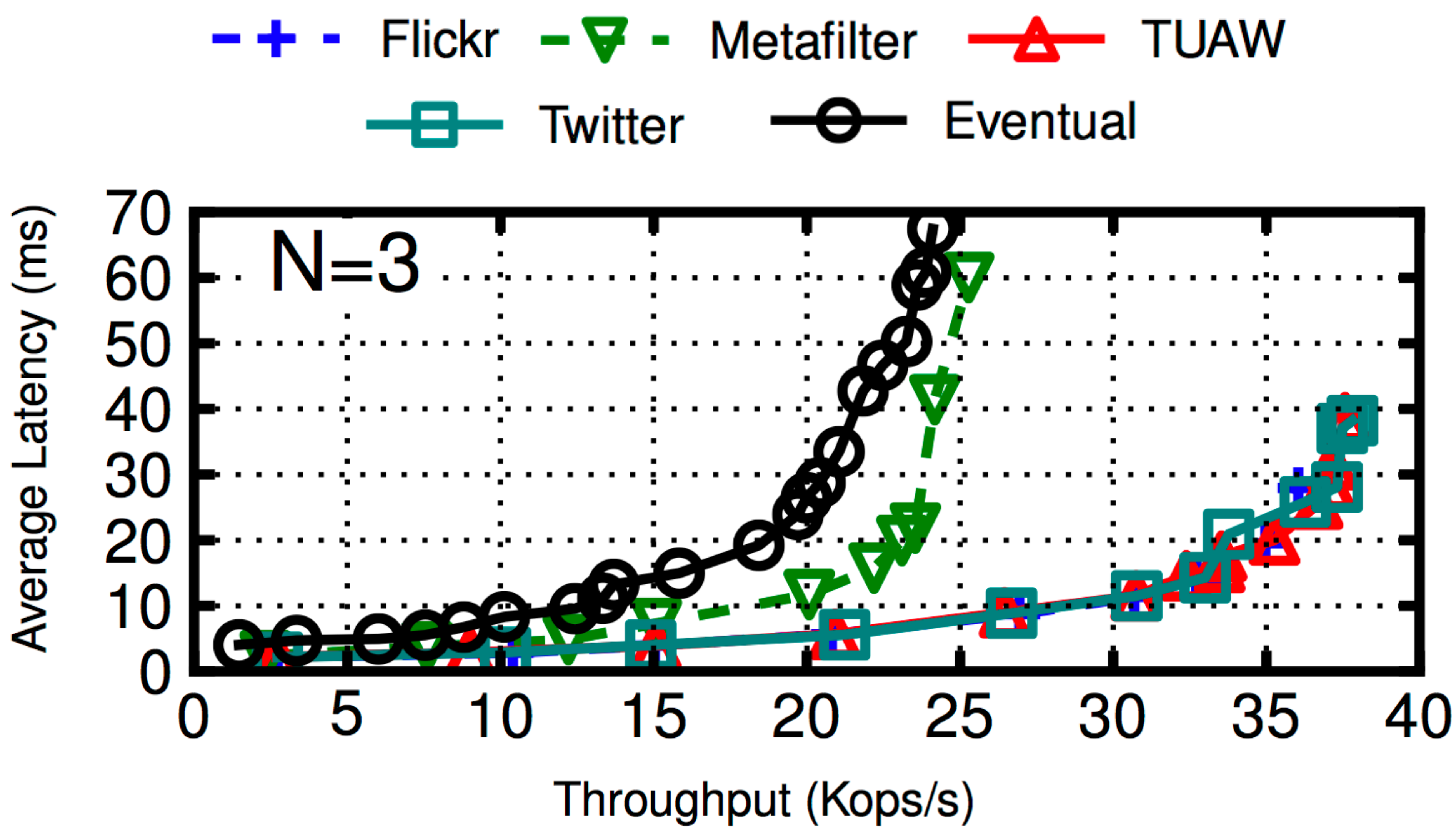
Continuous trade-off space between dependency resolution depth and fast-path latency hit

Strategy 2: Fetch dependencies asynchronously

Strategy 2: Fetch dependencies asynchronously

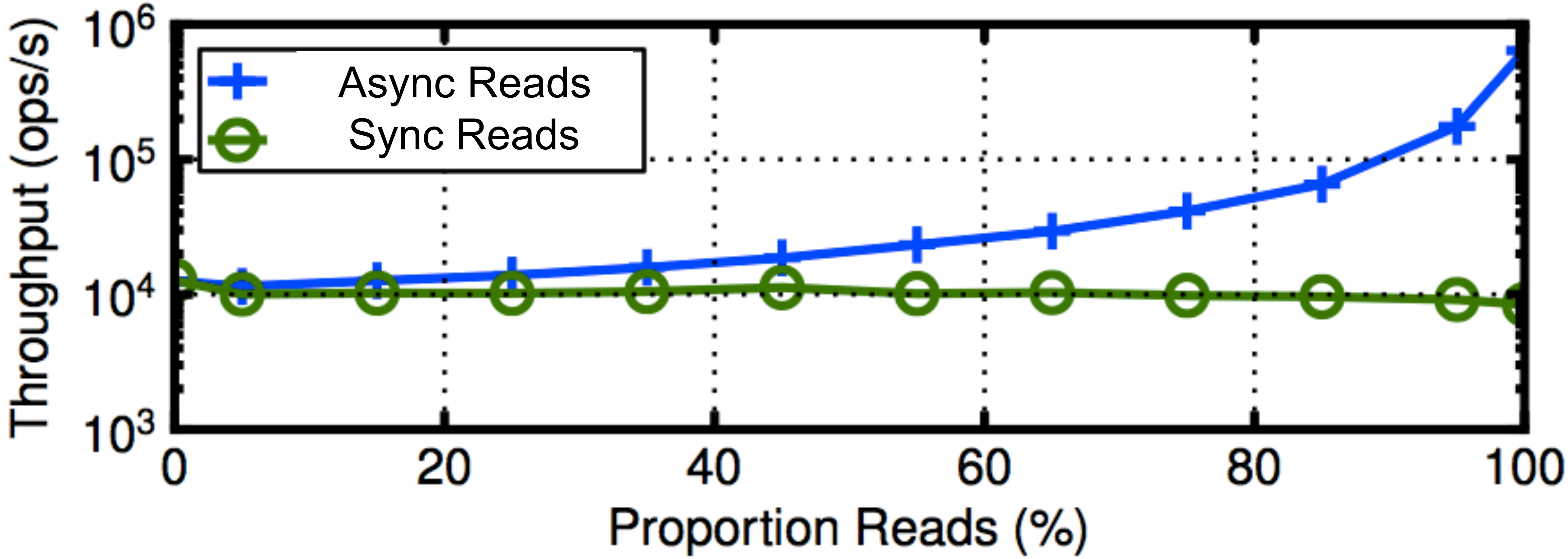


Strategy 2: Fetch dependencies asynchronously

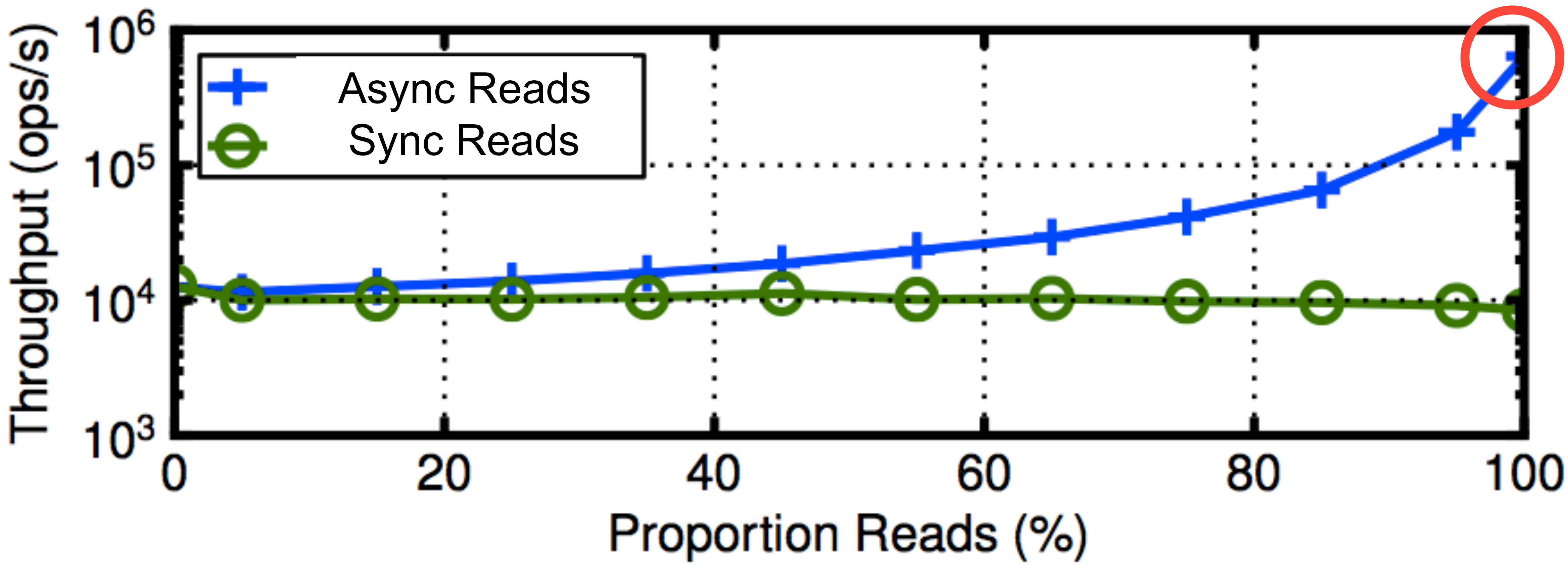


Throughput **exceeds eventual** configuration

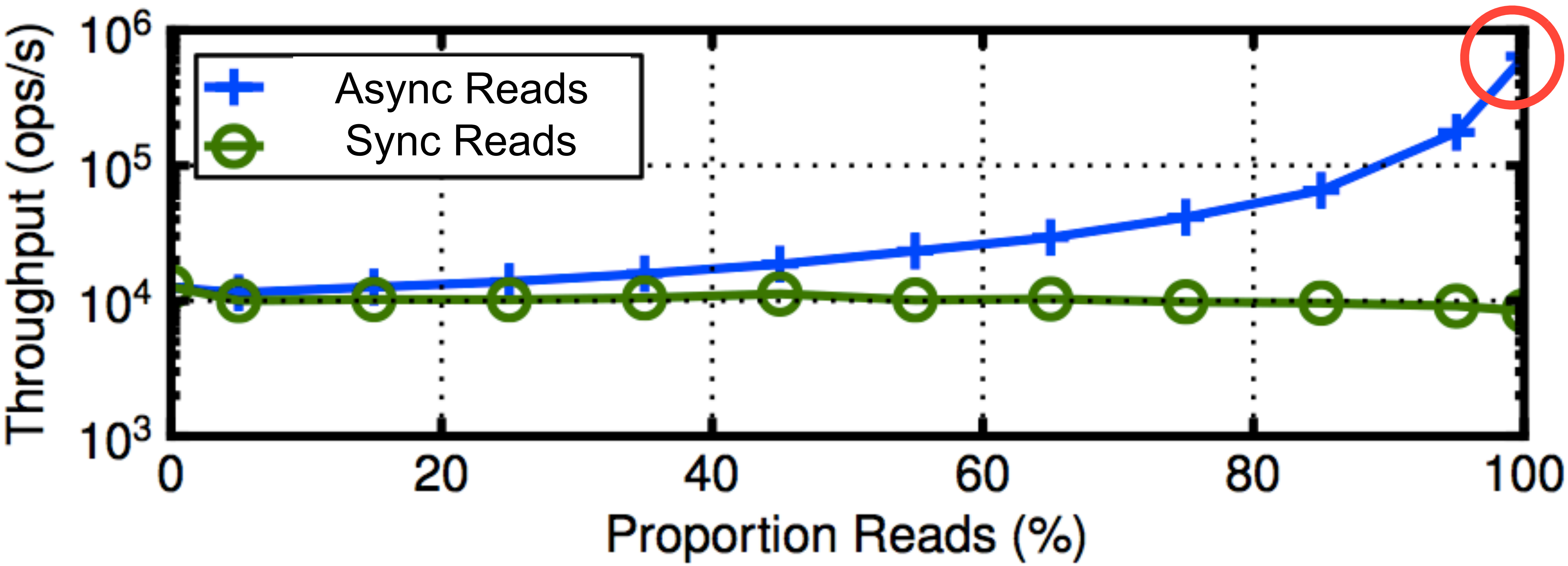
Still causally consistent, more stale reads



Async Reads
Sync Reads
Throughput (ops/s)
10^6
10^5
10^4
10^3
0
20
40
60
80
100
Proportion Reads (%)



Reading from cache is fast; linear speedup



Reading from cache is fast; linear speedup

...but not reading most recent data...

...in this case, effectively a straw-man.

Lessons

Causal consistency is achievable without modifications to existing stores

represent and **control** ordering between updates

EC is "orderless" until convergence

trade-off between visibility and ordering

Lessons

Causal consistency is achievable without modifications to existing stores

represent and **control** ordering between updates

EC is "orderless" until convergence

trade-off between visibility and ordering

works well for workloads with small causal histories, good temporal locality

Rethinking the EC API

Uncontrolled overwrites **increased metadata** and **local storage** requirements

Clients had to **check causal dependencies** independently, with no aid from EC store

Rethinking the EC API

What if we eliminated overwrites?

via multi-versioning,

conditional updates

or immutability

Rethinking the EC API

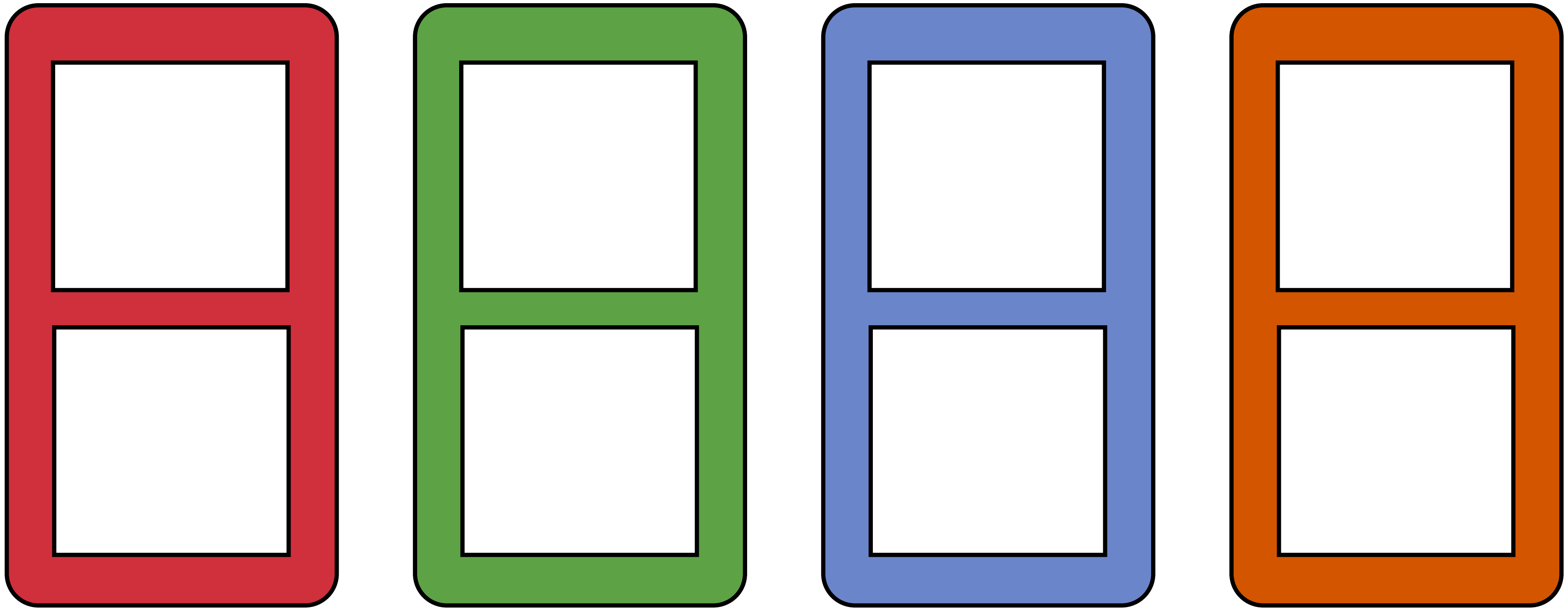
What if we eliminated overwrites?

via multi-versioning,
conditional updates
or immutability

No more overwritten histories
Decrease metadata
Still have to check for dependency arrivals

Rethinking the EC API

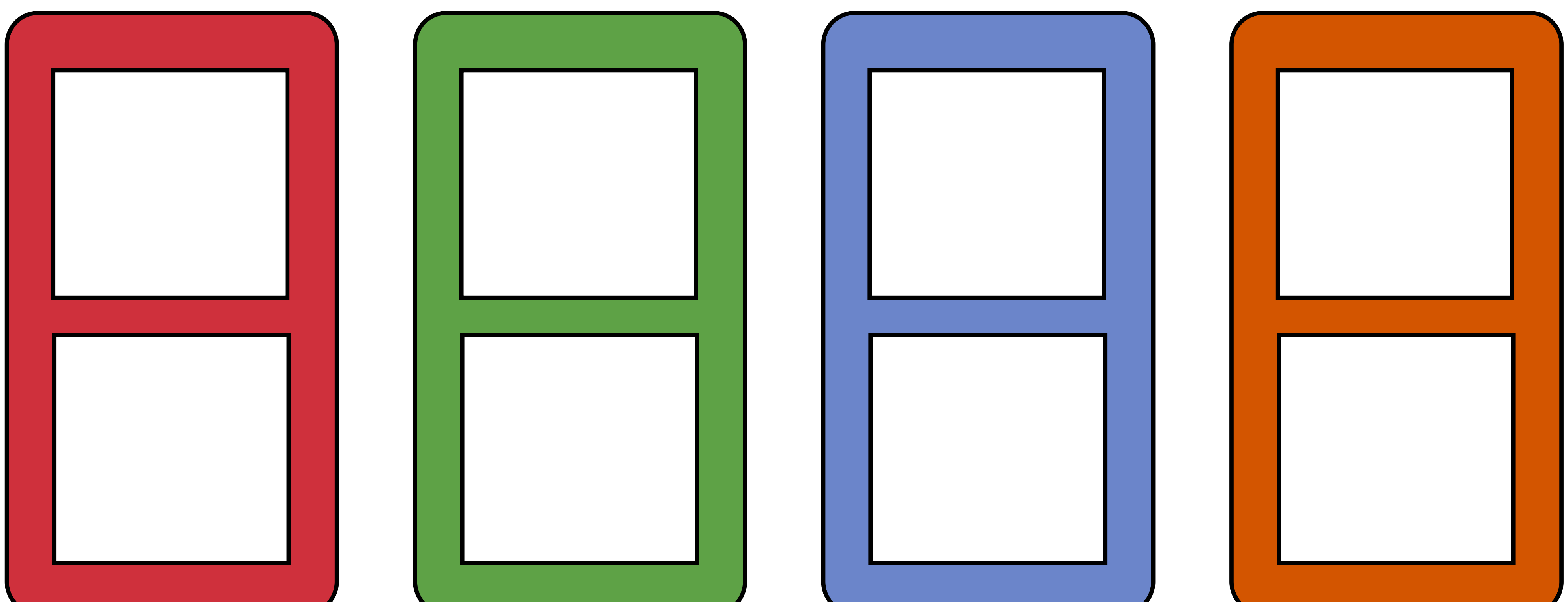
What if the EC store notified us when dependencies converged (arrived everywhere)?



Rethinking the EC API

What if the EC store notified us when dependencies converged (arrived everywhere)?

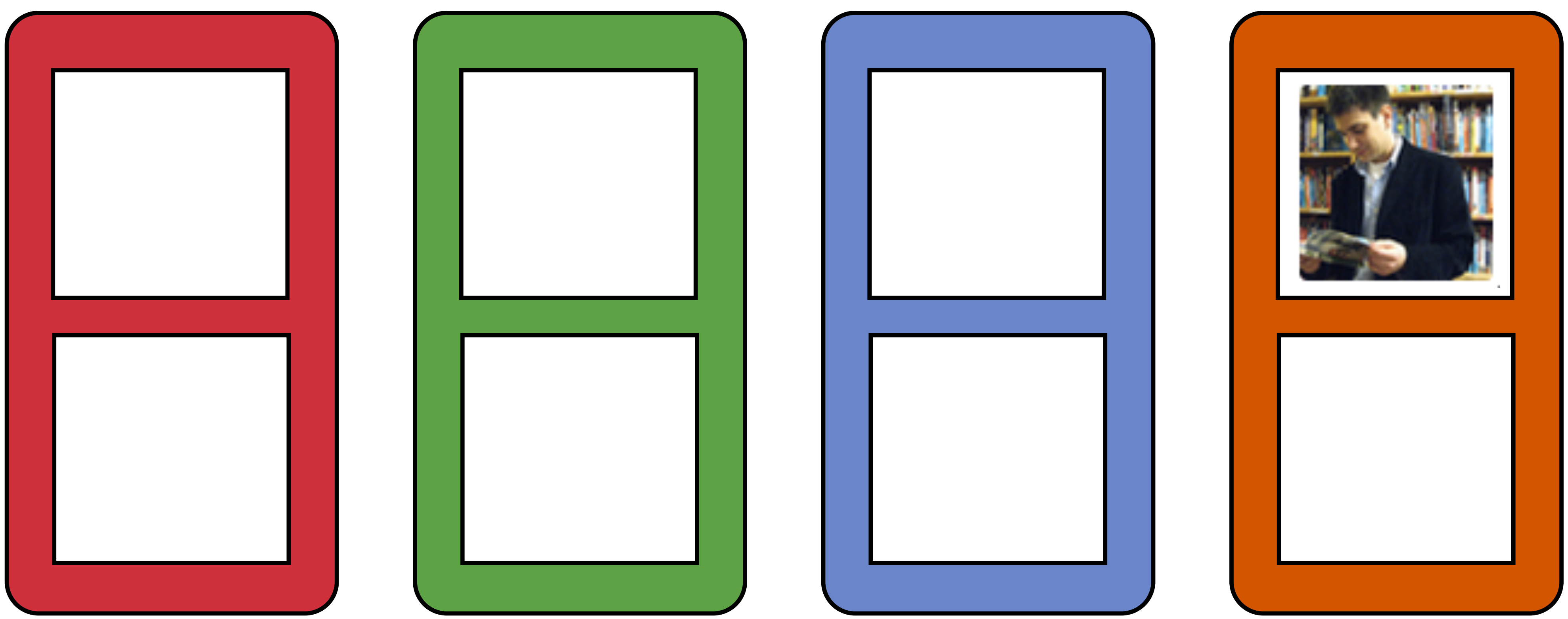
put( **after**  **converges)**



Rethinking the EC API

What if the EC store notified us when dependencies converged (arrived everywhere)?

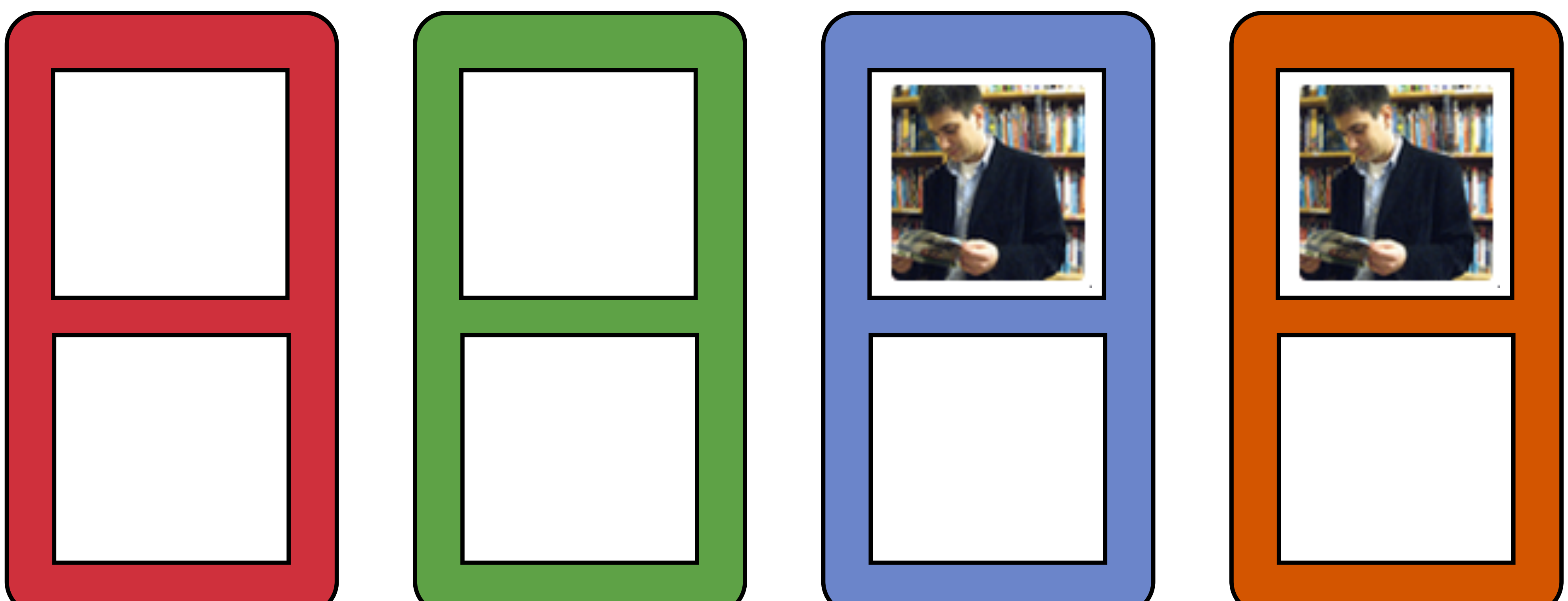
put( **after**  **converges)**



Rethinking the EC API

What if the EC store notified us when dependencies converged (arrived everywhere)?

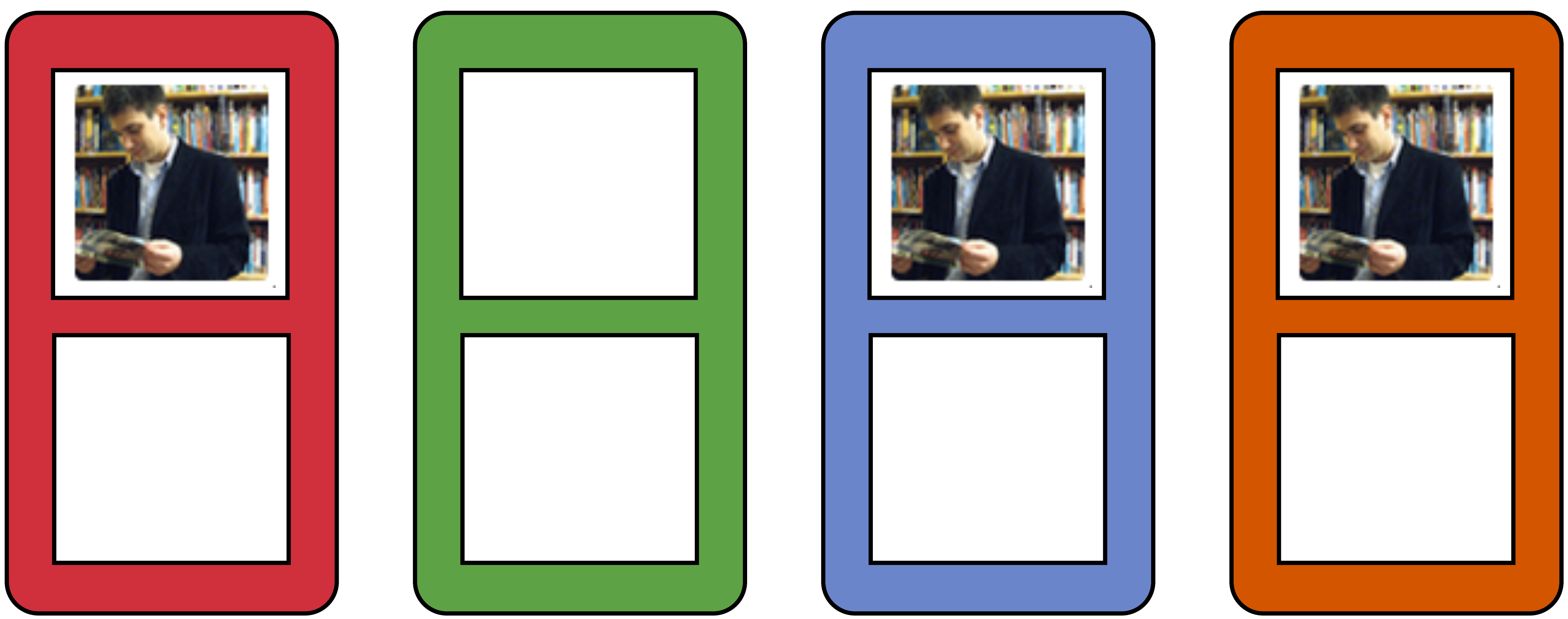
put( **after**  **converges)**



Rethinking the EC API

What if the EC store notified us when dependencies converged (arrived everywhere)?

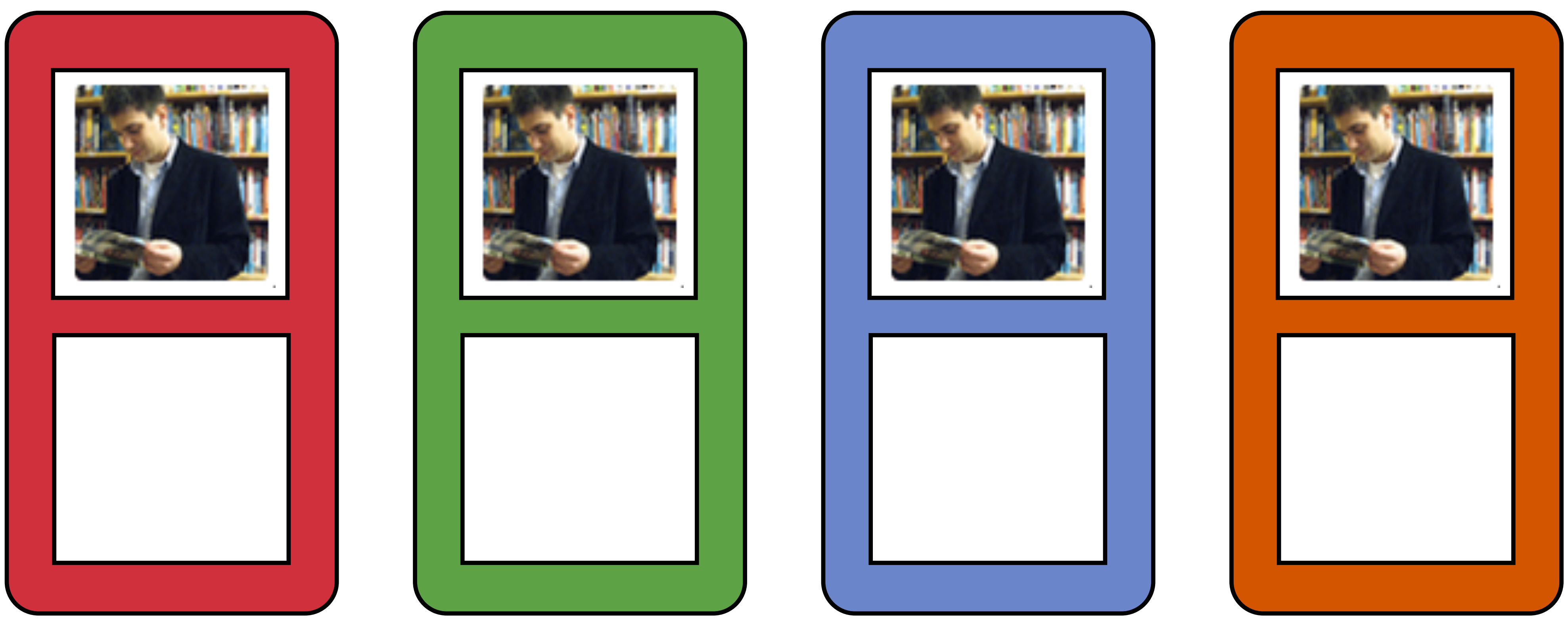
put( **after**  **converges)**



Rethinking the EC API

What if the EC store notified us when dependencies converged (arrived everywhere)?

put( **after**  **converges)**



Rethinking the EC API

What if the EC store notified us when dependencies converged (arrived everywhere)?

put( **after**  **converges)**

Rethinking the EC API

What if the EC store notified us when dependencies converged (arrived everywhere)?

put( **after**  **converges)**

Rethinking the EC API

What if the EC store notified us when dependencies converged (arrived everywhere)?

put( after  converges)

Rethinking the EC API

What if the EC store notified us when dependencies converged (arrived everywhere)?

put( **after**  **converges)**

Rethinking the EC API

What if the EC store notified us when dependencies converged (arrived everywhere)?

Wait to place writes in shared EC store until dependencies have converged

No need for metadata
No need for additional checks
Ensure durability with client-local EC storage

Reduces Metadata
No Dependency Checks

	Multi-versioning or Conditional Update
Reduces Metadata	YES
No Dependency Checks	NO

	Multi-versioning or Conditional Update	***Stable Callback***
Reduces Metadata	YES	YES
No Dependency Checks	NO	YES

	Multi-versioning or Conditional Update	***Stable Callback***
Reduces Metadata	YES	YES
No Dependency Checks	NO	YES

...not (yet) common to all stores

Data Store	***Multi-versioning or Conditional Update***	***Stable Callback***
Amazon DynamoDB	YES	NO
Amazon S3	NO	NO
Amazon SimpleDB	YES	NO
Amazon Dynamo	YES	NO
Cloudant Data Layer	YES	NO
Google App Engine	YES	NO
Apache Cassandra	NO	NO
Apache CouchDB	YES	NO
Basho Riak	YES	NO
LinkedIn Voldemort	YES	NO
MongoDB	YES	NO
Yahoo! PNUTS	YES	NO

Rethinking the EC API

Our extreme approach (unmodified EC store) definitely impeded efficiency (but is portable)

Opportunities to better define surgical improvements to API for future stores/shims!

Bolt-on Causal Consistency

Modular, "bolt-on" architecture cleanly separates **safety** and **liveness**

upgraded EC (all liveness) to causal consistency, preserving HA, low latency, liveness

Challenges: overwrites, managing causal order

Bolt-on Causal Consistency

Modular, “bolt-on” architecture cleanly separates **safety** and **liveness**

upgraded EC (all liveness) to causal consistency, preserving HA, low latency, liveness

Challenges: overwrites, managing causal order

large design space:

took an extreme here, but:
room for exploration in EC API
bolt-on transactions?

(Some) Related Work

- S3 DB [SIGMOD 2008]: foundational prior work building on EC stores, not causally consistent, not HA (e.g., RYW implementation), AWS-dependent (e.g., assumes queues)
- 28msec architecture [SIGMOD Record 2009]: like SIGMOD 2008, treat EC stores as cheap storage
- Cloudy [VLDB 2010]: layered approach to data management, partitioning, load balancing, messaging in middleware; larger focus: extensible query model, storage format, routing, etc.
- G-Store [SoCC 2010]: provide client and middleware implementation of entity-grouped linearizable transaction support
- Bermbach et al. middleware [IC2E 2013]: provides read-your-writes guarantees with caching
- Causal Consistency: Bayou [SOSP 1997], Lazy Replication [TOCS 1992], COPS [SOSP 2011], Eiger [NSDI 2013], ChainReaction [EuroSys 2013], Swift [INRIA] are all custom solutions for causal memory [Ga Tech 1993] (inspired by Lamport [CACM 1978])